

CTOS

CTOS[®]

Operating System

**Concepts
Manual**

Volume 2

UNISYS

UNISYS

CTOS[®]
Operating System
Concepts
Manual
Volume 2

Copyright © 1991, 1992 Unisys Corporation
All Rights Reserved
Unisys is a trademark of Unisys Corporation



Printed on recycled paper

CTOS III 1.0
CTOS II 3.4
CTOS I 3.4
CTOS/XE 3.4
Development Utilities 12.2
Standard Software 12.2
Video Access Method 4.0

July 1992

Printed in USA
4360 2630-000

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information or software material, including direct, indirect, special or consequential damages.

You should be careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

RESTRICTED RIGHTS LEGEND. Use, reproduction, or disclosure is subject to the restrictions set forth in DFARS 252.227-7013 and FAR 52.227-14 for commercial computer software.

Convergent, Convergent Technologies, CTOS, and NGEN are registered trademarks of Convergent Technologies, Inc.

OFIS is a registered trademark of Unisys Corporation.

BTOS is a trademark of Unisys Corporation.

Context Manager, Context Manager/VM, CT-Net, CTOS/VM, Document Designer, Generic Print System, PC Emulator, Print Manager, Shared Resource Processor, Solution Designer, SRP, Voice/Data Services, Voice Processor, and X-Bus are trademarks of Convergent Technologies, Inc.

OS/2 is a registered trademark of International Business Machines Corporation. Intel is a registered trademark of Intel Corporation. Lotus is a registered trademark of Lotus Development Corporation.

Page Status

Page	Issue
v thru xxxvi	Original
28-1 through 28-3	Original
28-4	Blank
29-1 through 29-8	Original
30-1 through 30-55	Original
30-56	Blank
31-1 through 31-21	Original
31-22	Blank
32-1 through 32-18	Original
33-1 through 33-28	Original
34-1 through 34-12	Original
35-1 through 35-16	Original
36-1 through 36-23	Original
36-24	Blank
37-1 through 37-8	Original
38-1 through 38-17	Original
38-18	Blank
39-1 through 39-22	Original
40-1 through 40-32	Original
41-1 through 41-10	Original
42-1 through 42-9	Original
42-10	Blank
43-1	Original
43-2	Blank
44-1 through 44-4	Original
45-1 through 45-35	Original
45-36	Blank
A-1 through A-5	Original
A-6	Blank
Glossary-1 through 69	Original
Glossary-70	Blank
Index-1 through 41	Original
Index-42	Blank

Contents

About This Manual	xxxvii
--------------------------------	--------

Section 1. Introduction to CTOS

What Is CTOS?	1-1
CTOS Foundation	1-2
Event-Driven, Priority-Ordered Process Scheduling	1-3
Message-Based Operation	1-3
Multiprogramming	1-3
Multitasking (Multithreading)	1-4
Nationalization	1-4
Networking Built-In	1-4
Protected Mode Enhancements	1-4
Caching	1-5
Extended Native Language Support (ENLS)	1-5
Multiple and International Keyboard Support	1-5
Protected Mode Operation	1-6
Real Mode Application Support	1-6
SCSI Management	1-6
Variable Partitions With Code Sharing Capability	1-7
CTOS III Enhancements	1-7
Demand Paging	1-7
Dynamic Link Libraries (DLLs)	1-7
Name Management	1-8
Semaphores	1-8

Section 2. Overview of Operating System Concepts

What is a Process?	2-1
What Constitutes the Kernel?	2-1
Event-Driven Priority-Ordered Scheduling	2-2
Interprocess Communication (IPC)	2-2
System Service Processes	2-3
System Service Filters	2-4
Inter-CPU Communication (ICC)	2-4

Contents

Command Interpreter: the Executive	2-4
File Management System	2-5
User-Written Device Handlers	2-5
Distributed Environment and Clustering	2-6
Local Resource-Sharing Networks (Clusters)	2-6
Network	2-6
Operating System Types	2-7
Workstation Operating Systems	2-8
Shared Resource Processor Systems	2-9
Programs and Run Files	2-10
Setting Up the Operating Environment	2-10
Partitions and User Numbers	2-11
Memory Management Styles	2-11
Multipartition Memory Management	2-12
Variable Partition Memory Management	2-12
Virtual Memory Management	2-13
Memory Organization at System Initialization	2-14
Variable Partition and Multipartition	
Operating Systems	2-14
Virtual Memory Operating Systems	2-16
Loading Applications	2-18
Bringing Applications into Memory	2-18
Managing Physical Memory	2-19
Swapping	2-19
Demand Paging	2-20
Virtual Code Management	2-22
Application Partition Memory Organization	2-22
Dynamic Linking	2-24
Code and Data Sharing	2-24

Section 3. Demand Paging

Demand Paging Overview	3-1
Demand Paging Terminology	3-2
Demand Paging: A Memory Management Style	3-3
Benefits of Demand Paging	3-4
Page Mapping	3-5
Allocating Linear Address Space	3-6
Oversubscribing the Physical Address Space	3-8
Replacing Pages	3-9

Cleaning Pages	3-11
Prefaulting Pages	3-11
Page Locking	3-12
Paging Service Components	3-12
Executing Real Mode Applications	3-13
Querying Paging Statistics	3-13
Tuning Paging Through System Configuration	3-14
Demand Paging Operations	3-15

Section 4. Using CTOS Operations

Introduction to Using CTOS Operations	4-1
Before You Begin Programming on CTOS	4-1
Naming Conventions	4-2
Programmatic Interface	4-2
Example CTOS Call in C Language	4-3
Operation Types	4-4
Object Module Procedures	4-5
System-Common Procedures	4-5
Kernel Primitives	4-6
Using the Request Procedural Interface	4-6
Using the Kernel Primitives	4-7
I/O Interface Levels	4-8
Addressing Memory	4-10
Logical Memory Address	4-12
Linear Memory Address	4-13
Physical Memory Address	4-13
Advantages to Protected Mode Memory Addressing	4-14
Extended Memory	4-14
Protection	4-14
Selecting Operations for Program Portability	4-15

Section 5. Program Management

What Is Program Management?	5-1
What Is a Program?	5-1
Linking a Program	5-3
Loading a Program Into Memory	5-4

Contents

Program Termination	5-5
Loading the Exit Run File	5-5
Notifying Other Programs of Termination	5-5
Deallocating System Resources	5-6
Error Handling	5-6
Program Management Operations	5-7
Error Handling	5-7
Normal Program Exit	5-8
 Section 6. Parameter Management	
What is Parameter Management?	6-1
Program Using Parameter Management	6-1
Parameters	6-2
Structures and Operations Overview	6-3
Application System Control Block (ASCB)	6-3
Variable Length Parameter Block (VLPB)	6-4
Querying Parameters In the VLPB	6-4
Example of a VLPB for the Rename Command	6-6
Operations for Constructing the VLPB	6-8
VLPB Structure	6-9
Parameter Management Operations	6-10
Constructing Parameters	6-10
Querying Parameters	6-10
 Section 7. Input/Output	
In This Section	7-1
Manual Sections Describing I/O	7-1
Device Independence Versus Dependence	7-3
I/O Facilities	7-3
 Section 8. Sequential Access Method	
What is the Sequential Access Method?	8-1
Customizing the Sequential Access Method	8-2
Byte Stream	8-3
Using a Byte Stream	8-3
Types of Byte Streams	8-4
Disk Byte Streams	8-5
Printer Byte Streams	8-5
Generic Print System Byte Streams	8-6
Pre-GPS Spooler Byte Streams	8-7

Keyboard Byte Streams	8-8
Communications Byte Streams	8-8
X.25 Byte Streams	8-9
Video Byte Streams	8-9
Sequential Access Byte Streams	8-9
Device/File Specifications	8-10
Device/File Specification Parsing	8-14
SAM Operations	8-15
Basic	8-15
Advanced	8-16

Section 9. Device Dependent SAM

What is Device-Dependent SAM?	9-1
Mapping to Device-Dependent Operations	9-1
Device-Specific Operations	9-2
Device-Dependent SAM Operations	9-4

Section 10. Video

What is the Video Facility?	10-1
Character-Map and Bit-Map Video	10-1
Video Attributes	10-2
Video Software	10-2
Using the SAM Operations	10-3
Using the Current Screen Setup	10-3
Using SAM Directly	10-4
Augmenting the SAM Operations	10-4
Special Characters in Video Byte Streams	10-5
Multibyte Escape Sequences	10-5
Controlling Screen Attributes	10-5
Controlling Character Attributes	10-6
Controlling Scrolling and Cursor Positioning	10-6
Dynamically Redirecting a Video Byte Stream	10-6
Automatically Pausing Between Full Frames	10-7
Miscellaneous Functions	10-7
Using QueryVidBs	10-8
Using the Video Access Method (VAM)	10-8
Using Video Display Management (VDM)	10-8
Reinitializing the Video Subsystem	10-9
Forms-Oriented Interaction	10-10
Advanced Text Processing	10-11

Workstation Video Capabilities	10-11
Character Cell	10-14
Character-map	10-15
Video Attributes	10-15
Font	10-15
Cursor	10-16
Video Refresh	10-16
Writing Portable Video Programs	10-16
Video Data Structures	10-17
Programming Using Color	10-17
Video Operations	10-18
VAM Operations	10-18
Video Requests	10-20
VDM Operations	10-20
Color Programming Operations	10-22
Direct Access to Video Data Structures	10-23

Section 11. Keyboard and I-Bus Management

What is Keyboard Management?	11-1
Keyboard Terminology	11-2
Keyboard Management Features	11-5
Keyboard Management Overview	11-7
Preprocessing	11-9
Postprocessing	11-10
Keyboard Modes	11-10
Unencoded Mode	11-11
Character mode	11-12
Other Keyboard Modes	11-12
Comparing the Keyboard Modes	11-13
Keyboard Hardware Protocols	11-13
Reading the Keyboard	11-14
Using ReadKbdInfo	11-14
Information Returned to the Requesting Application ...	11-15
Status Word Information	11-15
Type-Ahead Buffer	11-16
Writing to the Type-Ahead Buffer	11-17
Keyboard Data Block Organization	11-18
Finding a Key Definition	11-20
Translation Data Block Organization	11-20

Translating	11-22
Supporting Tables	11-22
Selecting the Translation Table to Use	11-23
Obtaining Other Keyboard Details	11-25
Contents of the Default K1 Translation Data Block	11-25
Decoding	11-26
Comparing Data Blocks	11-26
Emulating	11-28
Emulation Examples	11-28
Redefining Keys	11-30
Emulating LEDs	11-31
Customizing the Keyboard Data Blocks	11-31
Customizing the System Keyboard Data Blocks	11-32
Building a New System Keyboard File	11-32
Customizing Application-Specific Data Blocks	11-32
Dynamically Customizing Data Blocks for Application Use	11-33
If Your Application Uses NLS Keyboard Tables	11-33
Modifying Data Blocks	11-34
Posting Data Blocks	11-36
Reading Keyboard Data Blocks	11-36
Using ReadOsKbdTable	11-37
Using Byte Streams	11-37
Binary Data Block File to Local Memory	11-38
Data Block in an Object Module	11-39
Comparing Methods of Reading Data Blocks	11-39
Application Customization Examples	11-40
Multibyte Strings	11-40
Diacritics	11-40
Chords that Toggle	11-40
System Profile Keyboard	11-41
Matching Keyboards to Data Blocks	11-42
Application Profile Keyboard	11-43
Multiple Keyboard Profiles	11-43
Mapping Keyboard IDs	11-44
Setting Keyboard Options	11-45
Sticky Keys	11-45
Character Repeating	11-45

Contents

System Input Process	11-46
Playback Mode	11-47
Recording Mode	11-49
Submit File Escape Sequences	11-49
Read-Direct Escape Sequences	11-51
Action Key	11-52
At Program Termination	11-54
Keyboard and Video Independence	11-54
I-Bus Device Management	11-54
Keyboard and I-Bus Management Operations	11-56
Basic	11-56
Advanced	11-57
I-BUS Management	11-59
Magnetic Card Reader	11-59

Section 12. File Management

What is File Management?	12-1
Overview of File System Capabilities	12-1
Efficiency	12-1
Reliability	12-2
Convenience	12-3
Structured File Access Methods	12-3
Access to Local Files	12-4
File Specifications	12-4
Node	12-5
Volume	12-5
Directory	12-7
File	12-7
Password	12-8
Directory and File Specifications	12-9
Abbreviated Specifications	12-10
Automatic Volume Recognition	12-11
File Protection	12-11
Protection by Password	12-12
Volume Password	12-12
Directory Password	12-12
File Password	12-13
Device Password	12-13

Using a Password For Access	12-13
Protection by Protection Level	12-14
How Protection Levels Work	12-14
How The Operating System Validates	
Protection Levels	12-16
Protection by Volume Encryption	12-18
Creating and Accessing a File	12-20
Structured File Access Methods	12-20
Byte Streams	12-20
File Management Operations	12-20
Logical File Address	12-21
File Handle	12-22
From Creating to Deleting a File	12-22
Creating a File	12-22
Opening a File	12-24
Reading and Writing a File	12-25
Closing a File	12-26
Deleting a File	12-27
Local File System	12-28
LfsToMaster	12-29
Volume Control Structures	12-29
Volume Home Block	12-30
Allocation Bit Map and Bad Sector File	12-32
File Header Block	12-32
Disk Extent	12-32
Extension File Header Block	12-32
Master File Directory and Directories	12-33
System Directory	12-33
System Data Structures	12-34
User Control Block	12-34
Device Control Block	12-35
Building and Parsing File Specifications	12-35
Terms	12-36
Using The File Building and Parsing Operations	12-36
Layers of Access	12-37
Validating File Specifications	12-38
Syntax Errors	12-40
Wild Card Operations	12-41
\$ Directory	12-42

File Management Operations	12-43
Basic	12-43
Basic Utility Operations	12-43
File Attributes	12-44
Default Path	12-45
Directories	12-45
Long-Lived Files	12-46
File Handle Operations	12-46
Parsing Specifications	12-47
Asynchronous File I/O	12-49
Volume Data Structures	12-49
 Section 13. Disk Management	
What Is Disk Management?	13-1
Accessing a Disk Device	13-1
Device Specification and Password	13-2
Memory Disk	13-3
Cached Memory Disk	13-3
Disk Partitions	13-4
Disk Management Operations	13-5
 Section 14. Printing Management	
Generic Print System Components	14-1
Interface Considerations	14-2
 Section 15. Communications Programming	
What Is Communications Programming?	15-1
What SamC Is Used For	15-1
What Programs Use SamC	15-2
What Programs Cannot Use SamC	15-2
At the Device-Independent Interface Level	15-2
At the Device-Dependent Interface Level	15-3
Using the SamC Operations	15-4
Asynchronous Interface	15-4
The AcquireByteStreamC Operation (Low-Level Open)	15-5
Dynamically Changing Parameters	15-5
Querying and Setting Status Lines	15-5
The CheckForOperatorRestartC Operation	15-5
SamC Operations	15-6

Section 16. Serial Port Management

Access Below the Byte Stream Level (CommLine)	16-1
Serial Port Operations	16-2
Using InitCommLine	16-2
Using ResetCommLine	16-3
Using ChangeCommLineBaudRate	16-3
Using ReadCommLineStatus	16-4
Using WriteCommLineStatus	16-4
Serial Port Management Operations	16-5

Section 17. SRP Terminal Management

Program Access to Ports	17-1
SRP Terminal Management Operations	17-3

Section 18. SCSI Device Management

What is SCSI Management?	18-1
SCSI Management Terminology	18-3
Overview of the SCSI Manager's Capabilities	18-4
Convenience	18-4
Efficiency	18-5
Reliability	18-5
Creating a SCSI Path	18-5
Configuring a SCSI Manager	18-7
Defining a Unique Path	18-7
Specifying Path Parameters	18-8
Changing and Querying Path Parameters	18-9
Using the Path Handle	18-9
SCSI Access Modes	18-10
SCSI Passwords	18-11
File System and the SCSI Manager	18-12
SCSI Command Structure	18-14
Error Conditions and SCSI Sense Data	18-16
SCSI Manager Target Mode	18-20
Automatic Target Mode Functions	18-21
Explicitly Enabling Target Mode Functions	18-21
Managing Data Packets	18-22
Operating Methods for Target Mode Commands	18-22
Using a Target Check or Wait Operation	18-23

Contents

Example of Using the SCSI Manager	18-23
SCSI Management Operations	18-26
SCSI Paths	18-26
Basic SCSI I/O	18-27
Advanced SCSI	18-27
Target Mode	18-28
 Section 19. Generic Print Access Method	
What is the Generic Print Access Method?	19-1
 Section 20. Structured File Access Methods	
What is Structured File Access?	20-1
ISAM	20-2
RSAM	20-3
DAM	20-4
Hybrid Access Patterns	20-4
Modifying and Reading Data Files	20-5
Selecting a File Access Method	20-6
Disk Space	20-6
Access Time	20-7
Structured File Access Methods Operations	20-8
 Section 21. Indexed Sequential Access Method	
What is ISAM?	21-1
 Section 22. Record Sequential Access Method	
What is RSAM?	22-1
RSAM Files and Records	22-1
RSAM Working Area	22-2
RSAM Buffer	22-2
RSAM Operations	22-3
Basic	22-3
Advanced	22-3

Section 23. Direct Access Method

What is DAM?	23-1
DAM Files, Records, and Record Fragments	23-1
DAM Working Area	23-2
DAM Buffer	23-2
Buffer Size and Sequential Access	23-3
Buffer Management Modes	23-3
DAM Operations	23-4
Basic	23-4
Advanced	23-4

Section 24. Memory Management

What is Memory Management?	24-1
Memory Management Terminology	24-1
Partition Memory	24-2
Global Linear Address Space	24-3
Segments	24-3
Addressing a Byte in a Segment	24-3
Segment Limit and Huge Segments	24-4
Code, Static Data, and Dynamic Data Segments	24-4
Expand Up and Expand Down Segments	24-5
From Source Modules to Program in Memory	24-6
Models of Segmentation	24-6
Run Files	24-6
Application Partition Memory Organization	24-8
Allocating and Deallocating Partition Memory	24-9
Order of Deallocating Partition Memory	24-10
Using Long-Lived Memory	24-10
Using Short-Lived Memory	24-11
Using Global Linear Address Space	24-11
Obtaining Memory Addressed by a Single Selector	24-12
Obtaining Memory Addressed by a Selector Sequence	24-12
Memory Management Operations	24-13
Short-Lived Memory	24-13
Long-Lived Memory	24-14
Short-Lived and Long-Lived Memory	24-14
I/O Management	24-15
Selector Management	24-15
Global Linear Address Space Management	24-17

Section 25. Cacher

What is the Cacher?	25-1
Cacher Terminology	25-2
Why Use the Cacher?	25-3
Cacher Data Structures	25-4
Cache Pool Descriptor	25-4
Cache Entry Descriptor	25-5
Initializing a Cache	25-5
Allocating Cache Memory	25-5
Calculating Number of Cache Entries	25-6
Formatting Cache Memory	25-6
Hashing	25-8
Obtaining a Cache Entry	25-9
Cache Optimizations	25-10
Chaining Cache Buffers	25-10
Deferring a Cache Hit	25-10
Preventing Buffer Stealing	25-10
Releasing a Cache Entry	25-11
Conditions Before Release of an Entry	25-11
At Release of an Entry	25-12
Flushing Cache Entries	25-12
Obtaining Cache Status	25-12
Obtaining Cache Statistics	25-13
Closing a Cache Pool	25-13
Cacher Operations	25-14

Section 26. Utility Operations

What are Utility operations?	26-1
Accessing Resources in Disk Files	26-2
What Are Resources?	26-2
Why Resource Management?	26-3
Resource Work Area	26-4
Stepping On Files	26-5
Process Overview	26-5
Starting And Ending a Resource Session	26-7
Initializing Resource Access	26-8
Saving to Disk	26-9
Copying Resources	26-11
Copying One Resource at a Time	26-11
Using a Base RWA	26-11
Copying Multiple Resources	26-12

Copying a Resource From a CTOS Data File	26-13
Deleting Resources	26-13
Copying The Nonresource Portion of a Run File	26-14
Accessing Resources in Memory	26-14
Comparing Strings	26-14
Using StringsEqual and NlsULCmpB	26-15
Using WildCardMatch	26-15
Handling Nationalized Strings	26-15
Customizing the User Interface	26-16
Directing Data to a Byte Stream	26-16
Handling Commands	26-18
Managing Names	26-18
Classes	26-18
Configuring the Name Heap	26-19
Using the Name Management Operations	26-19
Manipulating Error Messages	26-20
Obtaining the System Date and Time	26-21
Parsing Configuration Files	26-23
Parsing User Configuration Files	26-23
Parsing Standard Configuration Files	26-24
Parsing Nonstandard Configuration Files	26-25
Using Unique Workstation Hardware IDs	26-27
Performing Miscellaneous Tasks	26-27
Building a Single-Line Text Editor	26-27
Comparing Logical Addresses	26-28
Copying Files	26-28
Determining Monitor Resolution	26-28
Writing Records to the System Log File	26-28
Informing User of Waiting Mail	26-28
Utility Operations	26-29
Accessing Resources in Disk Files	26-29
Accessing Resources in Memory	26-30
Comparing Strings	26-31
Customizing the User Interface	26-32
Directing Data to a Byte Stream	26-33
Handling Commands	26-35
Managing Names	26-35
Manipulating Error Messages	26-35
Obtaining the System Date and Time	26-36
Obtaining Versions of System Services	26-38
Parsing Configuration Files	26-40
Using Workstation Hardware IDs	26-42
Performing Miscellaneous Tasks	26-43

Section 27. System Definitions

System Definitions in This Section	27-1
Methods of Obtaining System Information	27-9
System Definition Operations	27-10
Cluster Management	27-10
Disk Management	27-10
File Management	27-10
Operating System	27-10
User Name Management	27-13
Video	27-13

Section 28. Multiprogramming

Benefits of Multiprogramming	28-1
Simple Multiprogramming Event Sequence	28-1
Multiprogramming Subjects	28-2

Section 29. Process Management

How is a Process Perceived?	29-1
By an End User	29-1
By a Programmer	29-1
By the Operating System	29-2
Process Management	29-2
Context of a Process	29-2
Process Priorities and Scheduling	29-3
Process States	29-4
Process Management Operations	29-7

Section 30. Interprocess Communication

What is Interprocess Communication?	30-1
An IPC Example	30-1
What Really Happens	30-2
Request Procedural Interface	30-2
System Service	30-3
IPC Summary	30-3
Other IPC Applications	30-4
Communication Within an Application Partition	30-4
Communication Between Application Partitions	30-5
Synchronization	30-6
Resource Management	30-7

Why Understand IPC?	30-9
Request Codes	30-9
Interprocess Communication (IPC) Components	30-10
Kernel Primitives for Sending a Message	30-12
Request and Respond	30-12
Send	30-15
ForwardRequest and RequestDirect	30-15
Kernel Primitives for Receiving a Message	30-16
Wait	30-16
Check	30-17
The Exchange	30-17
Types of Exchanges	30-18
Exchange Allocation	30-18
Sending a Message to an Exchange	30-19
Waiting for a Message at an Exchange	30-21
Exchange Queues	30-22
The Message	30-23
Request Block Format	30-24
Standard Request Block Header	30-25
Control Information	30-26
Routing Code	30-27
Request Data Item	30-27
Response Data Item	30-27
Example Request Block	30-28
Accessing System Services	30-30
Using the Request Procedural Interface	30-30
Using the Kernel Primitives Directly	30-31
Cluster/Network Communication	30-32
Cluster Configuration	30-33
Workstation Agent	30-33
Master Agent	30-33
Extending Resource Sharing	30-34
Routing by Handle	30-34
Routing by Specification	30-41
Rules for Routing by Specification	30-41
Expanding Specifications	30-42
Routing Code	30-43
Routing Requests	30-45
Network Routing	30-46
Shared Resource Processor Routing	30-47
Exchange Routing	30-49

Filter Process	30-51
Interprocess Communication Summary	30-52
Interprocess Communication Operations	30-54

Section 31. Semaphores

What is a Semaphore?	31-1
Who Uses Semaphores?	31-2
Semaphore Terminology	31-2
Semaphore Types	31-4
System and RAM Semaphores	31-4
Mutual Exclusion Semaphores	31-5
Critical Section Semaphores	31-5
Noncritical Semaphores	31-6
Signaling Semaphores	31-6
Opening a System Semaphore	31-6
Providing Exclusive Access to a Resource	31-7
Noncritical Semaphores	31-7
Locking a Noncritical Semaphore	31-7
Clearing a Noncritical Semaphore	31-8
Critical Section Semaphores	31-8
Using the Critical Section Operations	31-9
Terminating and Suspending Process	31-10
Mutual Exclusion on CTOS	31-10
Behind-the-Scenes Use of a Semaphore	31-11
Implementing a Simple Mutual Exclusion Semaphore ..	31-11
Signaling and Synchronizing Processes	31-12
Using the Signaling Semaphore Operations	31-12
The Producer/Consumer Model	31-13
CTOS Primitive Solutions to the Producer/Consumer Model	31-15
Using RAM Semaphores	31-16
RAM Semaphore Variable	31-17
Guidelines to Using RAM Semaphores	31-18
Semaphore Operations	31-20
System Semaphores	31-20
Mutual Exclusion	31-20
Critical Section	31-20
Signaling and Synchronization	31-20

Section 32. Inter-CPU Communication

What is Inter-CPU Communication?	32-1
ICC Terminology	32-2
Slot Numbers	32-2
Shared Resource Processor Routing Types	32-3
Blocks	32-5
CPU Description Table	32-6
ICC Segment	32-6
Ring Queues of Block Type Buffers	32-6
Bus Addresses	32-6
Ring Queues of Requests and Responses	32-7
Buffer Ownership Table	32-7
Buffer Wait Tables	32-7
Doorbell Interrupt	32-7
Interboard Routing	32-8
Sending a Request	32-8
Receiving a Request	32-11
Sending a Response	32-11
Receiving a Response	32-12
Sending and Receiving Messages	32-12
Mediating Buffer Wait Conditions	32-14
ICC Operations	32-18

Section 33. System Services Management

What Is System Services Management?	33-1
System Service Terminology	33-1
Overview of Operation	33-2
Built-In System Services	33-5
Dynamically Installable System Services	33-6
Request Routing Table	33-6
System Service Package	33-7
Requests	33-7
The System Service	33-8
Guidelines for Writing a System Service	33-8
Initialization and Conversion to a System Service	33-8
System Service Main Program	33-13
Restrictions and Requirements of Operation	33-14
Guidelines for Defining Requests	33-14
Creating Loadable Request Files	33-16

System Requests	33-17
Termination and Abort Requests	33-18
Termination Request to the File System	33-19
Swapping Requests	33-19
Change User Number Requests	33-20
Converting to a Multi-Instance Service	33-21
Filters	33-22
Types of Filters	33-22
Replacement	33-22
One-Way Pass Through	33-22
Two-Way Pass Through	33-24
System Requests for Filters	33-25
Use of Filters	33-25
Results of Not Serving Swapping Requests	33-25
Deinstallation of a System Service	33-26
System Service Operations	33-28
Basic Requests Used by all System Services	33-28
System Requests	33-28

Section 34. System-Common Services Management

What is a System-Common Service?	34-1
System-Common Service Model Overview	34-2
Compared to Request-Based System Services	34-3
Similarities	34-4
Differences	34-5
Choosing a System Service to Write	34-7
System-Common Service Writing Guidelines	34-8
Serving Requests	34-8
Serving a Deinstallation Request	34-9
How to Write a System-Common Service	34-9
Calling the System-Common Service	34-11
To Eliminate Linking	34-11
System-Common Service Operations	34-12

Section 35. Extended System Services

What are Extended System Services?	35-1
Extended System Services Operations	35-2
Asynchronous System Service	35-2
CD-ROM Service	35-4
Command Access Service	35-6
Mouse Services	35-6

Performance Statistics Service	35-9
Queue Manager	35-10
Sequential Access Service	35-12
Spooler Management	35-14
Voice/Data Services	35-14

Section 36. Partitions and Partition Management

What Is Partition Management?	36-1
Partitions	36-2
Types	36-2
Fixed or Variable	36-2
Partition Components	36-3
In-Memory Relationships	36-3
Local Descriptor Table	36-5
User Structure	36-5
Program Code	36-7
Short-Lived and Long-Lived Memory	36-7
User Number	36-8
Partition Organization In System Memory	36-9
At System Initialization	36-9
Single Application Partition In Memory	36-11
Multiple Application Partitions In Memory	36-12
Partition Swapping	36-13
Creating a Partition	36-15
Loading a Program	36-16
Terminating a Program	36-16
Removing a Partition	36-17
Obtaining Partition Status	36-17
Communicating Between Application Partitions	36-17
Application Partition With Multiple Run Files	36-18
Partition Management Operations	36-20
Basic Partition Management Operations	36-20
Swapping	36-21
Partition Creating Under Program Control	36-22
Partition Loading Under Program Control	36-22
Loading Additional Tasks	36-23

Section 37. Timer Management

System Timers	37-1
Realtime Clock	37-1
Programmable Interval Timer	37-1

Using the Timer Management Operations	37-2
Using Delay and Doze	37-2
Using ShortDelay	37-3
Realtime Clock	37-3
Timing a Single Interval	37-4
Repetitive Timing	37-5
Programmable Interval Timer	37-6
Timer Management Operations	37-8
Delay	37-8
Realtime Clock	37-8
Programmable Interval Timer	37-8

Section 38. Virtual Code Management

What is Virtual Code Management?	38-1
Overlays	38-2
Writing or Retrofitting Overlay Programs	38-2
Model Overview	38-3
Data Structures	38-4
Overlay Zone Header	38-6
StaticsDesc	38-6
Return Overlay Descriptors	38-7
ProcInfoNonres	38-7
Protected Mode Operation	38-8
Real Mode Operation	38-9
Intercepting Calls	38-9
Intercepting Returns	38-10
Importance of Call/Return Conventions	38-13
Calls to Procedural Addresses	38-13
Adjusting Addresses	38-14
Virtual Code Management Operations	38-16
Basic	38-16
Advanced	38-16

Section 39. Dynamic Link Libraries

What are Dynamic Link Libraries (DLLs)?	39-1
Dynamic Link Library Terminology	39-2
DLLs Compared to Other CTOS Services	39-3
Statically Linked Object Modules	39-4
DLL Procedures	39-4
Comparison of DLLs to System-Common Services	39-4
Request-Based System Services	39-6

How Dynamic Linking Works	39-6
Impact on Loading	39-8
Mapping Linktime to Runtime Addresses	39-9
Module Definition File	39-10
Using the Module Definition Utility	39-11
Using Resources	39-11
Calling DLL Procedures	39-11
Sharing Data Among DLL Clients	39-11
General Guidelines for Writing DLLs	39-12
Observing Caveats	39-13
Defining the DLL and Client Modules	39-14
Executing Initialization Procedures	39-14
Terminating Clients	39-15
Configuring Your System to use DLLs	39-16
Handling Errors	39-16
Setting Up DLL Search Paths	39-17
Defining DLL Segments	39-17
Assigning LDT Selectors	39-17
Summary of DLL Segments	39-19
Special Use of Instance Segments	39-20
Runtime Dynamic Linking	39-20
Requests to Perform Runtime Linking	39-20
DLLs: In Conclusion	39-21
Dynamic Link Library Operations	39-22

Section 40. Interrupt Handlers

Interrupt Handling Terminology	40-1
External Interrupt Handling Model	40-5
Device Handling	40-5
Device Handler Process	40-7
Device Interrupt Handler	40-7
Controlling When External Interrupts Occur	40-8
The Interrupt Flag	40-8
The Programmable Interrupt Controller	40-9
Pending and Lost Interrupts	40-11
Nonmaskable Interrupts	40-12
Operating System Interrupt Handler Styles	40-12
CRIHs and CMIHs	40-15
Guidelines for Writing a CRIH	40-16
Guidelines for Writing a CMIH	40-18

RIHs and MIHs	40-19
Guidelines for Writing an RIH	40-20
Guidelines for Writing an MIH	40-22
Examples of External Interrupt Handlers	40-22
Parallel Port Interrupt Handlers	40-22
X-BUS Interrupt Handlers	40-23
PseudoInterrupts	40-24
Internal Interrupts	40-25
Software Interrupts	40-25
Program Exceptions	40-25
Faults	40-26
Trap Handlers	40-27
Packaging of Interrupt Handlers	40-28
Application Program	40-29
System Service	40-29
Interrupt Handler Operations	40-30

Section 41. X-Bus Management

What is an X-BUS?	41-1
Locating the Base Address	41-1
Dynamically Obtaining the Base I/O Address	41-2
Computing the Module Base Address	41-2
X-BUS Module/Processor Memory Access	41-3
Accessing X-BUS Module Memory	41-3
Using X-Bus Operations to Access Memory	41-3
Specifying a Window Size	41-4
Accessing Modules In Protected Mode	41-5
Accessing Modules In Real Mode	41-5
X-BUS DMA	41-6
Communication and Start-Up Protocols	41-6
XBIS	41-7
X-BUS Interrupts	41-8
X-Bus Management Operations	41-9

Section 42. Bus Address Management

What is Bus Address Management?	42-1
Why Bus Addresses?	42-1
Bus Addresses on an SRP	42-3
Who Uses Bus Addresses?	42-5

Using DMA Buffers	42-6
Piecemealing the DMA Buffer	42-7
Deallocating the DMA Buffer	42-7
Bus Address Management Operations	42-8
 Section 43. Configuration Management	
System Administrative Actions	43-1
Programmer Actions	43-1
 Section 44. Cluster Management	
What is Cluster Management?	44-1
Cluster Environment	44-1
Status	44-1
Polling	44-2
Roll Call	44-2
Repoll	44-2
Request Routing Across the Cluster	44-3
Cluster Management Operations	44-4
 Section 45. Native Language Support	
What is Native Language Support?	45-1
Extended Native Language Support	45-1
Message Files	45-2
NLS Terminology	45-2
How to Take Advantage of NLS	45-2
NLS Tables	45-3
Detailed NLS Table Descriptions	45-7
Keyboard Mapping	45-7
File System Case	45-8
Lowercase to Uppercase	45-8
Video Byte Streams Text	45-8
Uppercase To Lowercase	45-8
Keycap Legends	45-9
Date And Time Formats	45-9
Number and Currency Formats	45-10
Date Name Translations	45-10
Collating Sequence	45-10
Character Class	45-12
Yes or No Strings	45-13
Special Characters	45-13

Contents

Keyboard Chords	45-13
Multibyte Escape Keys	45-13
NLS Strings	45-14
ClusterShare Keyboard	45-14
ClusterShare Keyboard Extended Codes	45-15
ClusterShare Video Translation	45-15
ClusterShare Keyboard Key Post Values	45-15
OFIS Spreadsheet	45-15
Context Manager	45-16
Gengo Date Formats	45-16
Using the NLS Tables	45-17
NLS Tables Loaded at System Initialization	45-17
Alternate NLS Table Sets	45-17
Internationalization	45-18
Extending Internationalization	45-19
ENLS Character Processing	45-20
ENLS String Processing	45-21
ENLS Line/Form Drawing	45-21
Updating Applications	45-22
Localization	45-23
Customizing the NLS Tables	45-23
Linking With the Appropriate ENLS Library	45-23
Other System Customization Requirements	45-24
Message File Facility	45-24
Creating and Editing Message Files	45-24
Message File Operation Sets	45-25
Using Macros With Messages	45-26
Native Language Support Operations	45-28
NLS Utility	45-28
ENLS Utility	45-30
Message File	45-33
 Appendix A. Operating System Features	 A-1
 Glossary	 1

Figures

2-1.	Variable Partiton and Multipartition Systems	2-15
2-2.	Virtual Memory Operating Systems	2-17
2-3.	Application Partition and Free Memory	2-18
2-4.	Memory Organization Under Partition Management	2-20
2-5.	Mapping Pages to Physical Memory Frames	2-21
2-6.	Memory Organization of an Application Partition	2-23
3-1.	Linear Address Space	3-7
3-2.	Oversubscribing Physical Memory	3-8
3-3.	Clock Algorithm	3-10
4-1.	Interface Levels	4-9
4-2.	Memory Address Translations	4-11
5-1.	From Source Modules to Program in Memory	5-2
6-1.	Executive Variable Length Parameter Block	6-5
6-2.	Filled-in Variable Length Parameter Block	6-7
7-1.	Interface Levels	7-2
11-1.	Keyboard Management Overview	11-8
11-2.	General Data Block Layout	11-19
11-3.	Translation Data Block Format	11-21
11-4.	Comparing Data Blocks	11-27
11-5.	NlsKbd.sys Format	11-35
11-6.	System Input Process	11-47
11-7.	Escape Sequence for Entering User Data	11-51
12-1.	Effects of Volume Encryption	12-19
12-2.	Volume Control Structures	12-31

Contents

17-1.	Ports/Access Methods Relationship	17-2
18-1.	Sample SCSI Configuration	18-6
18-2.	SCSI Manager Example	18-24
24-1.	Expand Up and Expand Down Segments	24-5
24-2.	From Source Modules to Program in Memory	24-7
24-3.	Application Partition Memory Organization	24-8
25-1.	Formatting Cache Memory	25-7
26-1.	Adding Resources to a Run File	26-6
26-2.	Resource Descriptor Table	26-9
26-3.	Name Heap	26-19
29-1.	Process States	29-5
30-1.	Client and System Service Interaction	30-2
30-2.	Processing Flow	30-3
30-3.	Communication Between Processes	30-4
30-4.	How IPC Is Used with ICMS	30-5
30-5.	Synchronization	30-6
30-6.	Buffer Management	30-8
30-7.	Request Primitive	30-13
30-8.	Respond Primitive	30-14
30-9.	Send Primitive	30-15
30-10.	Wait Primitive	30-16
30-11.	Sending a Message to an Exchange	30-20
30-12.	Waiting for a Message at an Exchange	30-21
30-13.	Messages Queued at an Exchange	30-22
30-14.	Processes Queued at an Exchange	30-23
30-15.	Request Block for the Write Operation	30-28
30-16.	Requesting to Open a File	30-37
30-17.	Responding With a File Handle	30-38
30-18.	Using the File Handle	30-40
30-19.	Network Routing	30-46
30-20.	Shared Resource Processor Routing	30-48
30-21.	Exchange Routing	30-50
30-22.	Filter Process Interaction	30-51
30-23.	Interprocess Communication Summary	30-52

31-1.	Semaphore Types	31-4
31-2.	Producer/Consumer Model Using Semaphores	31-14
31-3.	Producer/Consumer Model Using Send/Wait	31-15
31-4.	Producer/Consumer Model Using Request/Respond	31-16
31-5.	RAM Semaphore Variable Structure	31-17
32-1.	Z-Block	32-10
32-2.	ICC Interaction	32-13
32-3.	Board A's Wait Exchange Table	32-15
32-4.	Board B's Y-Block Wait Flags Table	32-15
32-5.	Wait Exchange and Wait-Satisfied Flags tables	32-16
33-1.	Client and System Service	33-3
33-2.	Processing Flow	33-5
33-3.	Request Routing Table Fields	33-7
33-4.	Before Conversion to a System Service	33-9
33-5.	Conversion to a System Service	33-12
33-6.	System Service Program Model	33-13
33-7.	One-Way Pass-Through Filter	33-23
33-8.	Two-Way Pass-Through Filter	33-24
34-1.	Request-Based Compared to System-Common	34-3
36-1.	Hypersegments in Protected Mode	36-4
36-2.	Memory Organization at System Initialization	36-10
36-3.	Single Application Partition in Memory	36-11
36-4.	Multiple Application Partitions in Memory	36-13
36-5.	Swapping	36-15
36-6.	Multiple Run Files in an Application Partition	36-19
38-1.	Virtual Code Facility Data Structures	38-5
38-2.	Stub Structure	38-8
38-3.	Tracing the Stack	38-11
39-1.	Execution Models	39-5
39-2.	Imports	39-8
39-3.	Assigning LDT Selectors	39-18

Contents

40-1.	Interrupt Hierarchy	40-4
40-2.	Device Handler	40-6
40-3.	Interrupt Nesting	40-11
40-4.	Interrupt Handler Styles	40-14
40-5.	CRIHs and CMIHs	40-15
40-6.	User-Written CRIH Summary	40-17
40-7.	User-Written CMIH Summary	40-19
40-8.	RIHs and MIHs	40-20
40-9.	User-Written RIH Summary	40-21
40-10.	User-Written MIH Summary	40-22
42-1.	Bus Address Types	42-2
42-2.	Using Bus Addresses on an SRP Board	42-4

Tables

ATM-1.	Common Prefixes	xlvi
ATM-2.	Common Roots	xlvii
2-1.	Workstation Operating System Features	2-8
2-2.	SRP Processor Board Features	2-9
3-1.	Demand Paging Terms	3-2
10-1.	Workstation Video Capabilities	10-12
10-2.	B24 and B27 Workstation Video Capabilities	10-13
10-3.	Character Cell Size	10-14
11-1.	Keyboard Terms	11-2
11-2.	Submit File Escape Sequence Permitted Codes	11-50
11-3.	Action Key Combinations	11-53
12-1.	Protection Levels	12-15
12-2.	Bit Number Designations for Protection Level	12-16
12-3.	Syntax Errors	12-41
18-1.	SCSI Device Management Terms	18-3
18-2.	Information Transfer Phases	18-14
18-3.	Sense Keys	18-19
24-1.	Memory Management Terms	24-1
25-1.	Cacher Terms	25-2
27-1.	System Structures	27-1
29-1.	Process State Transition	29-6

Contents

30-1.	Request Code Levels	30-10
30-2.	Format of a Request Block Header	30-25
30-3.	Interpretation of Resource Handle Bits	30-36
30-4.	Specification Expansion	30-43
30-5.	RtCode Values for Request Routing	30-44
30-6.	RTCode Values for Specification Expansion	30-44
31-1.	Semaphore Terms	31-3
32-1.	Shared Resource Processor Routing Types	32-4
33-1.	RequestTemplate.txt Fields	33-15
33-2.	Creating a Loadable Request File	33-17
34-1.	System Service Comparison	34-7
39-1.	DLL Terms	39-2
39-2.	Service Types Compared	39-3
39-3.	DLL Segments	39-19
40-1.	Interrupt Handling Terms	40-2
45-1.	NLS Terms	45-2
45-2.	NLS Tables	45-4
45-3.	Number and Currency Formats Key Elements	45-11

Section 28

Multiprogramming

This section serves as an introduction to information that will become more important to you as you gain familiarity with the more immediate and practical operating system concepts described in previous sections. The sections that follow describe the multiprogramming capabilities of the operating system.

Benefits of Multiprogramming

Multiprogramming allows several programs to be in memory at once. In addition to independent execution scheduling, these programs are provided the ability to communicate with each other.

For example, it is possible for a program to communicate with

- Programs in other partitions
- Programs at other workstations within a cluster
- Other processors of a shared resource processor (SRP)
- Programs at remote nodes

The sections on multiprogramming describe those events, transparent to you, that allow multiprogramming to take place.

Simple Multiprogramming Event Sequence

In a multiprogramming environment, the following are just a few of the events that take place as a result of your program request:

- The request is communicated to the file system by means of interprocess communication (IPC) and Kernel primitives.

- The file system manages access to the specified file, performs the requested service (sends output to the file), and responds to your program by means of IPC and Kernel primitives.
- Underlying these events, process management is at work, scheduling the execution activities of your program, the file system, and all other system processes that are competing with each other for processing time.

Multiprogramming Subjects

The multiprogramming subjects and a reference to the section in which each is described are as follows:

- **Processes:** A process is an independent thread of execution along with the hardware context (that is, the processor registers) necessary to that thread. One or more processes are created each time a program is executed. Certain operations manipulate processes, allowing you to create, prioritize, and obtain information about them for future programming reference. (See the section entitled “Process Management.”)
- **Interprocess Communication (IPC):** IPC is the core to communication among different processes. The section entitled “Interprocess Communication” describes the IPC concepts of messages and exchanges and the relationships of these concepts to processes.
- **Semaphore management:** Semaphores are a complement to IPC but are limited to local use. (See the section entitled “Semaphores.”)
- **Inter-CPU Communication (ICC):** ICC is the method by which messages are passed between processor boards on a shared resource processor. (See the section entitled “Inter-CPU Communication.”)
- **System Services:** System services act as managers of resources that can be accessed by application programs or other system services. The section entitled “System Services Management” describes how request-based system services use IPC to service clients and includes guidelines for writing such services.

- **System-Common Services:** System-common services provide similar functionality to that provided by request-based system services (described in “System Services Management”). The main difference is that system-common services are not request based. The section entitled “System-Common Services Management” compares the two types of service and describes the circumstances under which one type might be the better choice to use. In addition, this section provides guidelines for writing system-common services.
- **Extended System Services:** The section entitled “Extended System Services” describes system services that can be installed to become part of the operating system. These include Mouse Services and Voice/Data Services. All the extended system services are described in detail in the *CTOS Programming Guide*.
- **Partitions and Partition Management:** In the section entitled “Partitions and Partition Management,” the components of a partition or user number are described in detail. This introduction leads to a discussion of the partition structures that support the execution of multiple programs at the same time. In addition, this section describes the operations used by a partition managing program to create and to remove partitions under its management.
- **Timer Management:** The section entitled “Timer Management” describes the realtime clock (RTC) and the programmable interval timer (PIT).

Section 29

Process Management

How is a Process Perceived?

A *process* is a single thread of program execution. It can be perceived from three points of view:

- The end user sees two processes on the Executive screen.
- The programmer creates additional processes to perform separate functions within a single program by making the appropriate operating system calls.
- The operating system schedules processes to use the processor.

By an End User

As an end user, you can actually see two processes at work in the Executive. When you type into an Executive form, the main program process accepts your keystrokes. At the same time, a second process updates the clock. Whether or not you type, the clock continues to be updated.

By a Programmer

As a programmer, you perceive a process in terms of how you create an additional process in a multiprocess program like the Executive.

When the Executive run file is loaded into memory, the main program starts executing. At some point, it calls `CreateProcess`, which starts up the clock process. Each process executes as a separate thread. Global variables allow the main program and the clock to share the same data.

Typically, processes share the same code but have separate stacks. The degree and method of data sharing are program-specific.

By the Operating System

The operating system kernel views the two Executive processes as units competing for processor time. It is the operating system's responsibility to manage use of the processor among all existing processes.

Process Management

The operating system's process management facility always allocates the processor to the highest *priority* process currently requesting it. In the Executive for example, the clock process runs at a higher priority than the process accepting user keystrokes to ensure that the clock gets updated.

Scheduling is driven by system *events*. Whenever an event, such as the completion of an I/O operation, makes a higher priority process eligible for execution, rescheduling occurs immediately.

This scheduling technique is called *event-driven priority scheduling*. It reduces overhead and provides for a more responsive system than techniques that are entirely time-based.

To give multiple programs with the same priority a fair share of system resources, processes with priorities in a predefined range are optionally subject to time slicing.

Context of a Process

The *context of a process* is the collection of all information about a process. The context has both hardware and software components.

The hardware context of a process consists of values to be loaded into processor registers when the process is scheduled for execution. This includes the registers that control the location of the process's stack.

The software context of a process consists of its default response exchange, the priority at which it is scheduled for execution, and the interrupt vectors pointing to software interrupt handlers that the program uses.

The root structure containing the combined hardware and software context of a process is a system data structure called the process control block (PCB).

When a process preempts another process of lower priority, the operating system performs a *context switch* by saving the hardware context of the preempted process in that process's PCB. In protected mode, the context switch is done largely by the processor chip; the context state (registers) are saved in the task state segment (TSS). (For details, see the Intel documentation listed in "Related Documentation.") When the process is rescheduled for execution, the operating system restores the content of the registers, thus permitting the process to continue as though it were never interrupted.

Process Priorities and Scheduling

The priority of a process indicates the process's importance relative to other processes and is assigned at process creation. Priorities range from a high of 0 to a low of 255. Priorities and their normal (and recommended) uses are as follows:

Priority	Use
0-9	Operating system
10-64	System services
65-254	Application programs
255	Null process (see below)

The scheduler maintains a queue of the processes that are eligible to execute on a priority basis.

Rescheduling occurs when a system event makes a process executable because it has a higher priority than the one currently executing. Examples of system events include an interrupt from a device controller, X-Bus module, timer, or realtime clock, or a message sent from another process.

In most cases, the time interval between events is determined by the duration of a typical I/O operation. A process can lose control involuntarily only to a process of higher priority.

If a system event causes a message to be sent to an exchange at which a higher priority process is waiting, the operating system performs a *context switch* and reallocates the processor to execute the higher priority process.

When a system event occurs that makes a process eligible to execute, that process receives control of the processor until one of the following occurs:

- Another process with a higher priority preempts its execution.
- It voluntarily relinquishes control of the processor usually by calling the kernel primitive, `Wait`.

If no other process has work to perform, the *null process* is given control of the processor. The null process, which is always ready-to-run, executes at priority 255, lower than any real process.

In real mode, the null process examines the checksum value the operating system creates for its code segment when it is bootstrapped. The null process ensures that the checksum is valid. A variance in checksums indicates that a program has modified code and results in the operating system crashing with status code 91 ("Operating system checksum error").

In protected mode, the null process exists only to simplify the algorithm of the operating system scheduler.

To give multiple programs with the same priority a fair share of system resources, processes with priorities in a predefined range are subject to time slicing. Processes within this range that have the same priority are executed in turn for intervals of 100 milliseconds each in repeating cycles. The priority range is a system build parameter.

Process States

A process can exist in one of four states: running, ready, waiting, and suspended.

A process is in the *running state* when the processor is actually executing its instructions. Only one process can be in the running state at a time. Any other ready-to-run processes are in the *ready state*. Any number of processes can be in the ready state at the same time.

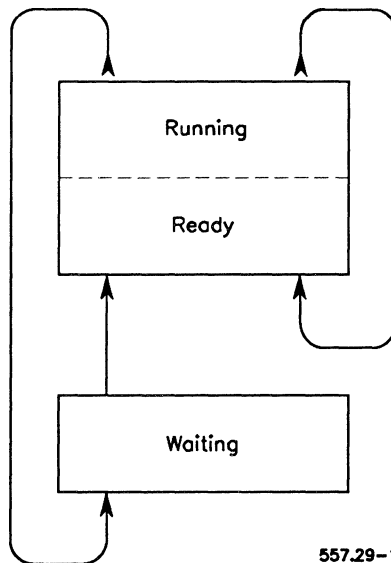
A process is in the *waiting state* when it waits at an exchange for a message to synchronize execution with other processes. A process enters the waiting state when it voluntarily issues the kernel primitive, *Wait*, which specifies an exchange where no messages are currently queued. Any number of processes can be waiting at a time.

As soon as the running process waits, the highest priority process in the ready state is placed in the running state.

If a process is *suspended*, it is also placed in either the ready or waiting states, but it is never placed in the running state. A process is suspended, for example, if it is subject to time slicing and its time slice runs out before it has completed executing.

The relationship among process states is shown in Figure 29-1. Table 29-1 describes the transitions between process states and the events causing the transitions.

Figure 29-1. Process States



557.29-1

Table 29-1. Process State Transition

Transition		Event
From	To	
Running	Waiting	A process executes a Wait but no messages are at the exchange.
Waiting	Ready	Another process sends a message to the exchange at which a process is waiting.
Running	Ready	A higher priority process leaves the waiting state and preempts this process.
Ready	Running	All higher priority processes enter the waiting state.

Process Management Operations

The process management operations are described below. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

CreateProcess

Creates a new process and schedules it for execution.

ChangePriority

Changes the priority of the calling process.

ChangeProcessPriority

Changes the priority of a process specified by the process ID.

SetDeltaPriority

Allows the dynamic changing of a process priority based on the memory partition it is executing in. *SetDeltaPriority* is used only by partition managing programs, such as the Context Manager.

SetDispMsw287

Directs the kernel to set the machine status word on every process context switch. (*SetDispMsw287* is used by software that manages or emulates the Math Coprocessor only on 80286 and higher microprocessor-based operating systems.)

RescheduleProcess

Moves a process in front of all other processes of the same priority on the run queue.

QueryProcessNumber

Allows a process to determine its own process ID.

NewProcess

Creates a new process and schedules it for execution. *NewProcess* differs from *CreateProcess* in that a user number and the initial values of all the registers may be specified, and a process ID is returned.

SuspendUser

Suspends all processes of the specified user number (partition).

UnsuspendUser

Releases all processes of the specified user number (partition) from the suspended state previously asserted with the SuspendUser operation.

SuspendProcess

Suspends the process identified by the specified process ID.

UnsuspendProcess

Releases the process identified by the specified process ID from the suspended state previously asserted with the SuspendProcess operation.

Section 30

Interprocess Communication

What is Interprocess Communication?

The *interprocess communication* (IPC) facility synchronizes process execution and information transmission between processes through the use of messages and exchanges. A process can communicate with another process in its own partition or in another partition.

Note: *In this section, the terms master and server are equivalent.*

An IPC Example

When you write a statement in your program, like

```
erc = OpenFile(&fh, &fileSpec, sizeof(fileSpec), 0, 0, 'mr');
```

your purpose might be to use the file handle (*fh*) returned to refer to the open file in a subsequent read or write statement. You assume that the statement will just work.

What you are actually doing is using the request procedural interface, one of the most common applications of IPC. A *request procedural interface* is an operating system routine that uses IPC to communicate the requested information in the statement you wrote to the file system service. IPC is used again to return the response information (file handle) from the file system to your program.

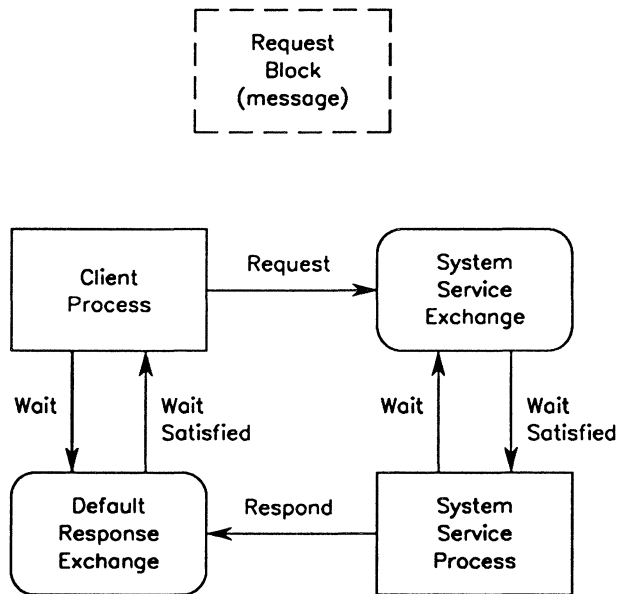
Your program is the *client*. Any program, including another system service, can be a client if it makes a request of a system service.

IPC provides the means by which a client and a system service communicate with each other. The communication is in the form of a special IPC message called a *request block*. The messenger facilitating the communication is the operating system kernel.

What Really Happens

When your program calls `OpenFile`, your program enters the operating system's request procedural interface routine. Figure 30-1 summarizes the interaction between the client and system service when this happens. The details of what is shown in the figure are described in the sections below.

Figure 30-1. Client and System Service Interaction



557.30-1

Request Procedural Interface

The request procedural interface

1. Builds the request block message for `OpenFile` in the client process's memory, copying information provided in the `OpenFile` statement.
2. Calls `Request` to route the request block to the system service exchange.
3. Places the client in a wait state at its default response exchange.

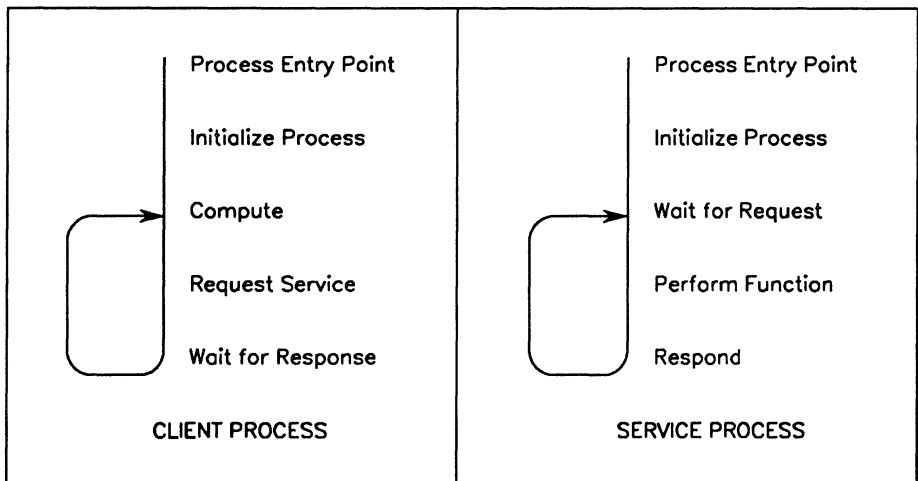
System Service

The system service waits at its service exchange for request blocks requesting its service. (See Figure 30-1.) Upon receipt of the request block, the system service verifies the information in the request block message. If the information is valid, the system service performs its service and answers the client's request by filling in the request block with its response and a normal status code (familiarly known as an `erc`). If the request is invalid, however, it places a status code indicating an error in the request block.

Upon completion of these functions, the system service calls `Respond` to route the request block back to the client's exchange. The client's wait is satisfied, and it is ready to execute its next instruction.

Figure 30-2 compares the processing flow of the client process to the system service process.

Figure 30-2. Processing Flow



557.30-2

IPC Summary

The request procedural interface uses IPC to send your request to the system service. You assume the information you requested will be available to use in your next program instruction.

The request procedural interface is a convenient way to access system services. It is compatible with BASIC, COBOL, FORTRAN-86, PL/M, C, and Pascal, as well as assembly language. For this reason, it is a common IPC application. IPC has other applications as well.

Other IPC Applications

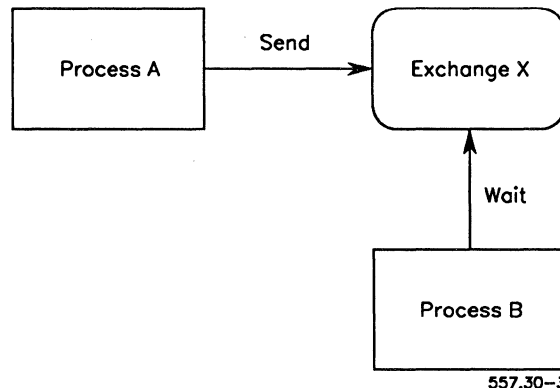
To a great extent, the power of the operating system results from its IPC facility. IPC supports three multiprogramming capabilities:

- Communication
- Synchronization
- Resource management

Communication Within an Application Partition

Communication, the most elementary interaction between processes, is the transmission of data from one process to another by means of an exchange. Figure 30-3 shows an example of communication. Process A and Process B are executing within the same application partition. Process A sends a message to Exchange X, and Process B waits for a message at that exchange. Note that the kernel primitive Send rather than Request is shown in the figure. Send is a special primitive used only by processes in the same partition. It is described in greater detail in “Kernel Primitives for Sending a Message.”

Figure 30-3. Communication Between Processes

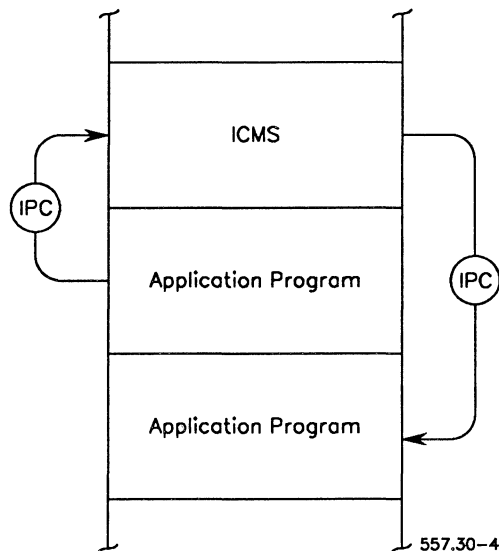


Communication Between Application Partitions

IPC is used in a special way by application programs that want to communicate with programs in other application partitions. This is done using the Intercontext Message Server (ICMS).

The requesting application program sends an IPC message to ICMS. ICMS, in turn, uses IPC to forward the message to the receiving program. If the receiving program is swapped out of memory, ICMS holds the message until the receiving program is resident again to accept it. Figure 30-4 shows how IPC is used with ICMS. (For details on ICMS, see your Context Manager manual.)

Figure 30-4. How IPC Is Used with ICMS

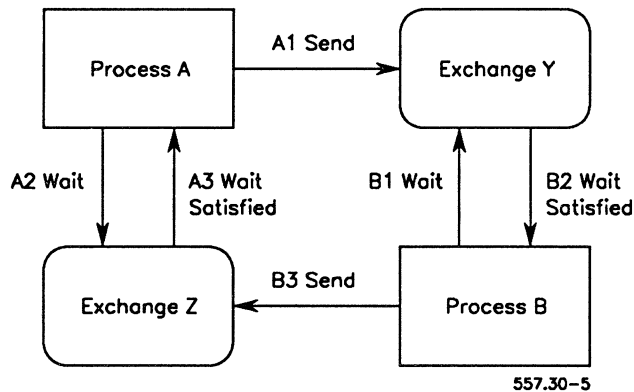


Synchronization

Synchronization is the means by which a process ensures that a second process has completed a particular item of work before the first process continues execution. Synchronization between processes and the transmission of data between processes usually occur simultaneously.

Figure 30-5 shows how Process A synchronizes with Process B in the same partition. In the figure, Process A sends a message to Exchange Y, requesting that Process B perform an item of work. Process A then waits at Exchange Z until Process B has completed the work. This synchronizes the continued execution of Process A with the completion of an item of work by Process B.

Figure 30-5. Synchronization



Resource Management

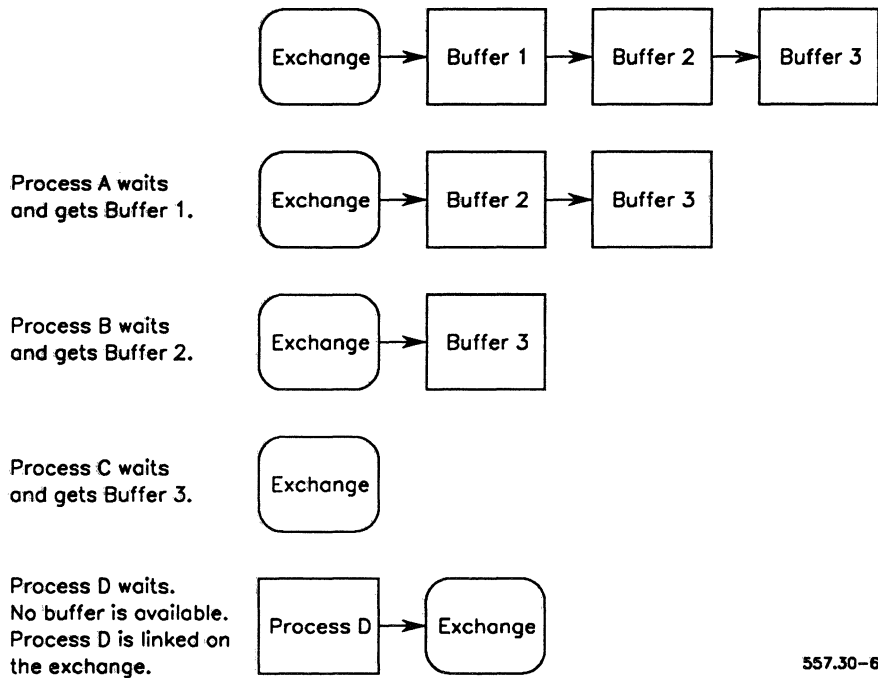
In a multiprogramming environment, *resource management* is the means of sharing resources among processes in a controlled way. For example, several processes may need to use the printer; however, only one process can use the printer at a particular time.

One way to control a resource is to establish a process to manage it. Then, only the managing process accesses the resource directly. Other processes access the resource indirectly by sending messages to the process that performs the chosen function. System service processes, which manage resources such as files, devices, and memory, are implemented by means of an analogous mechanism.

As an example of resource management, a pool of buffers may be available in a partition to be shared by processes performing I/O. When a buffer is not in use, a message indicating the number of the available buffer can be queued at an exchange set up to manage allocation of the buffers.

As shown in Figure 30-6, Process A waits at the exchange, picks up a message indicating that buffer 1 is available, and proceeds to use buffer 1. Process B waits at the exchange and is allocated buffer 2. Process C waits and is allocated buffer 3, the last available buffer. Then, when Process D waits, no buffer is available. Process D, therefore, must wait at the exchange until one of the processes using a buffer completes and returns a "buffer available" message back to the exchange.

Figure 30-6. Buffer Management



557.30-6

Why Understand IPC?

You can write statements like the `OpenFile` example (in “An IPC Example”), and IPC will work for you automatically.

At some point, however, you may want use some more advanced programming techniques to increase the efficiency of your program or to write your own system service. In these cases, you need to understand the mechanism behind IPC.

Request Codes

A *request code* is a 16 bit value that uniquely identifies a chosen system service operation. For example, the request code for the `OpenFile` operation is 4.

The request code is used in IPC both to route a request to the exchange of the appropriate system service and to specify which of its several functions the request refers to.

If you are writing a system service, you need to assign request codes to the requests you define to be performed by that service.

The operating system has a number of built-in system services of its own, such as the file system and keyboard. The request codes for operating system requests are listed in Appendix D in the *CTOS Procedural Interface Reference Manual*.

There are 64K possible request codes. They are divided into 16 byte request levels with 4K request codes at each level. To use the request procedural interface and validity checking structures, a request must be defined by a request code in an even-numbered level.

The request levels are shown in Table 30-1. Request codes in levels 0 through 9 are reserved for internal use. You must register request codes in levels A through D. Levels E and F are available without registration.

Note: *To reserve request codes in levels A through D, contact Unisys, Distributed Systems Division, San Jose Product Support.*

Level 0 must always be linked into the operating system, and it must contain definitions for all requests used by the operating system up to the point at which system initialization is complete. All other levels can be loaded at system boot time.

Initialization request structures also include tables for the system requests used for termination, abort, and swapping. System requests are defined by odd-level request codes. (For details, see “System Requests” in the section entitled “System Services Management.”)

Table 30-1. Request Code Levels

Level	Hexadecimal Values
0	0000 to 0FDF and FFE0 to FFFF
1	1000 to 1FFF
2	2000 to 2FFF
3	3000 to 3FFF
4	4000 to 4FFF
5	5000 to 5FFF
6	6000 to 6FFF
7	7000 to 7FFF
8	8000 to 8FFF
9	9000 to 9FFF
A	A000 to AFFF
B	B000 to BFFF
C	C000 to CFFF
D	D000 to DFFF
E	E000 to EFFF
F	F000 to FFDF

Interprocess Communication (IPC) Components

The basic components of IPC are listed below:

- Kernel primitives
- Exchanges
- Message (usually a request block)

The kernel primitives are used to send and receive messages.

Actually, the kernel primitives are inherently part of the kernel code: calling a primitive wakes up the kernel to perform some action.

The kernel acts as a messenger by sending messages to their appropriate destinations. When a system service waits to receive a message at its designated exchange, the kernel checks to see if any messages are there to be serviced.

The message conveys information and provides synchronization between processes. A 4 byte data item is communicated between processes. This data item is usually the memory address of a larger data structure that is called the *message*.

A message is actually sent to an *exchange* rather than directly to a process. An exchange can be thought of as serving the function of a post office where postal patrons (processes) go to mail (send) letters (messages) or pick up (wait or check for) letters.

In the same way that the postal patron drops a letter in the mail box and then walks away trusting that the letter will be delivered, a process sends a message and then continues executing without further regard.

A postal patron who is expecting an important letter can periodically go to the post office to check whether it has arrived. If the letter is especially important, the patron can wait in the post office for the letter to arrive.

A process has analogous mechanisms available when it expects to receive a message. It can check periodically whether a message is posted at (queued on) an exchange, or it can wait at the exchange for the arrival of a message. Because computers are much faster than the postal service, it is usually more appropriate to wait for a message than to check for its arrival.

A *request block* is a special type of message formatted according to specific conventions and used by all interprocess communications to the operating system.

Kernel Primitives for Sending a Message

The kernel primitives for sending a message include the following:

- Request
- Respond
- Send
- ForwardRequest
- RequestDirect

`RequestRemote` is an additional kernel primitive used to send a message to a remote processor on a shared resource processor. (The operation is described in “Shared Resource Processor Routing Types” in the section entitled “Inter-CPU Communication.”)

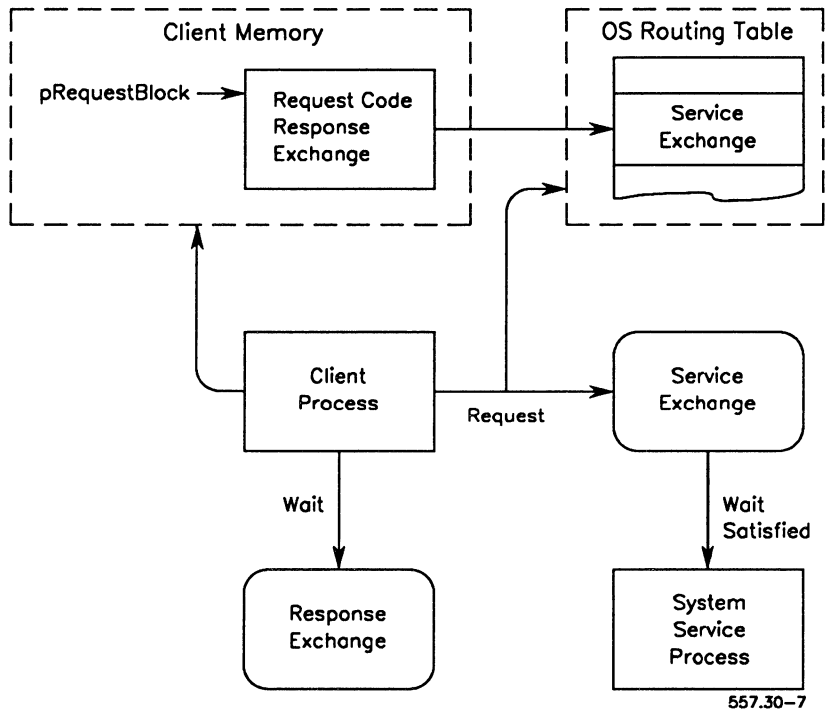
Request and Respond

`Request` is used by a client to direct a request to a system service. `Respond` is used by system services to respond back to the client. To provide a meaningful environment, each `Request` must be answered by a matching `Respond`.

The *Request* primitive directs a request for a system service from a client process to the service exchange of the system service process. (See Figure 30-7.) Before the `Request` is issued, the data required for the system service must be arranged in a request block in the client’s memory.

A request block is a special type of message formatted according to specific conventions and used by all IPCs to the operating system.

Figure 30-7. Request Primitive



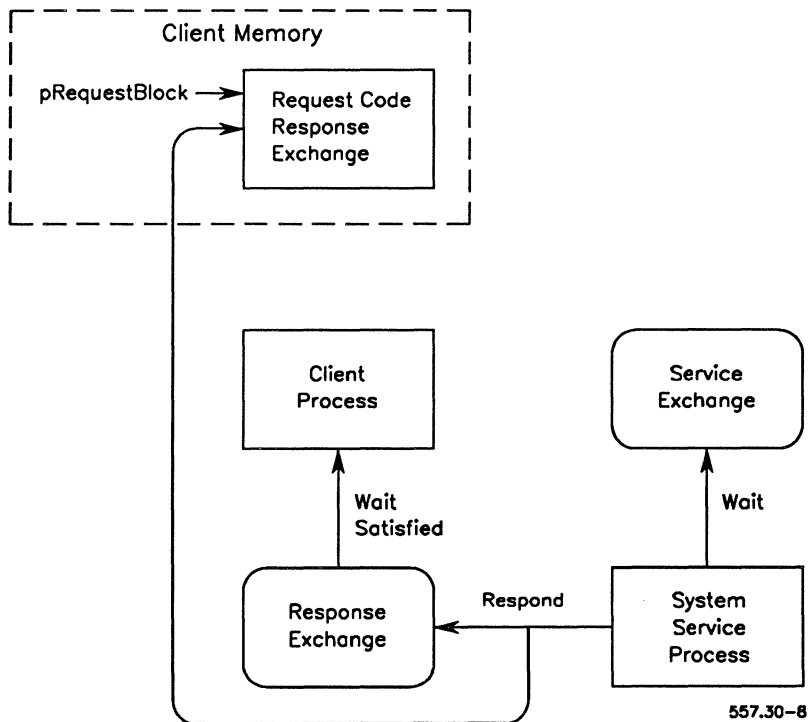
557.30-7

Request does not accept an identification of an exchange as a parameter. Instead, it uses the request code of the request block as an index into an operating system request routing table. The routing table resides in the System Image and contains the information the kernel needs to route the request to the appropriate service exchange. (For details on all the decisions the kernel makes to route a request based on this table, see “Routing Requests.”)

The *Respond* primitive is used by a system service process to send an answer to a request back to the response exchange of a client process. (See Figure 30-8.)

The only parameter to the Respond primitive is the memory address of the client's request block. That is, the system service must use (as a parameter to Respond) the same memory address that the client used as a parameter to Request. The exchange to which the response is directed is determined by the response exchange field of the request block.

Figure 30-8. Respond Primitive



Send

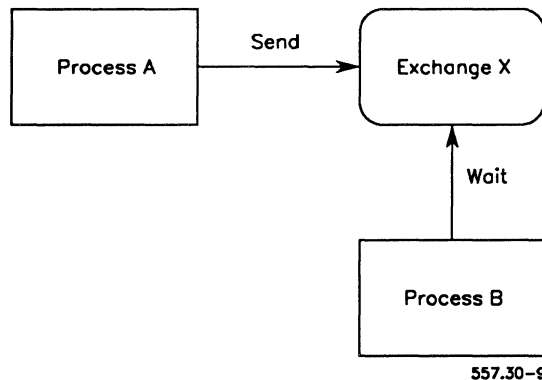
The *Send* primitive, unlike Request or Respond, accepts any 4 byte field as a parameter. This is usually, but not necessarily, the address of a message. Send does not require a formalized request block message, nor does it require a matching response.

Send should only be used by processes within the same partition (user number).

Note: *Send should not be used to send messages between programs in different application partitions. This is done in a special way by ICMS. For details, see “Communication Between Application Partitions.”*

Figure 30-9 shows how Send works in the communication between Process A and Process B in the same partition. Process A sends a message to Exchange X, and Process B waits for a message at that Exchange.

Figure 30-9. Send Primitive



ForwardRequest and RequestDirect

The ForwardRequest and the RequestDirect primitives are used by special types of system services called *filters*, which intercept messages destined to other system services. (See “Filter Process.” For details on how ForwardRequest and RequestDirect are used by filters, see “Types of Filters” in the section entitled “System Services Management.”)

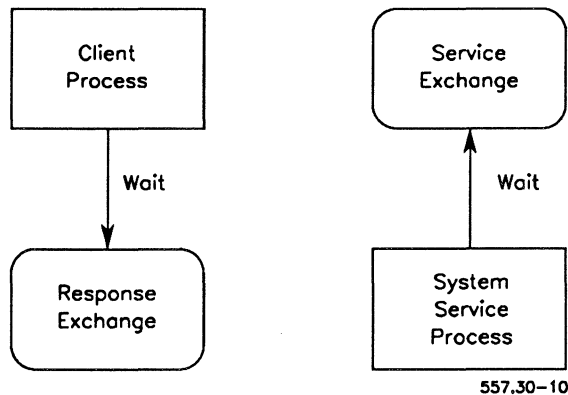
Kernel Primitives for Receiving a Message

There are two kernel primitives for receiving a message: *Wait* and *Check*. These primitives are described and illustrated in the paragraphs that follow.

Wait

The *Wait* primitive checks whether a message is queued at the specified exchange. System services wait at system service exchanges until their services are requested. Clients wait at response exchanges to synchronize their execution with the completion of a system service they request. (See Figure 30-10.)

Figure 30-10. Wait Primitive



In the context of all kernel primitives for sending messages except *Send* (that is, *Request*, *Respond*, *ForwardRequest*, *RequestDirect*, and *RequestRemote*), the message queued at the exchange is always a request block.

The details on how a process waits at an exchange to receive messages are described and illustrated in the following sections:

- “Sending a Message to an Exchange”
- “Waiting for a Message at an Exchange”

Check

The *Check* primitive checks whether a message is queued at the specified exchange. If one or more messages are queued, the message that was queued first is removed from the queue and its memory address is returned to the calling process. If no messages are queued, status code 14 (“No message available”) is returned.

Unlike the *Wait* primitive, the *Check* primitive never causes the calling process to wait.

(For details and examples of how to use the *Check* primitive, see “Accessing System Services.”)

The Exchange

A message is actually sent to an *exchange* rather than directly to a process. An exchange functions as a message center.

An exchange is referred to by a unique, 16 bit identifier. An exchange consists of two first-in, first-out queues. One is a queue of processes waiting for a message; the other is a queue of messages that are ready for processes to poll.

Note that either messages or processes (not both) can be queued at an exchange at any given time.

Only the address of the message, not the message itself, is sent to an exchange. This minimizes overhead. Therefore queueing a number of messages at the same exchange requires very little execution time or memory. IPC places no restriction on the size and content of the message. The receiving process must be programmed to use IPC to wait or to poll (check) for the availability of a message.

Types of Exchanges

A *response exchange* is the exchange at which the client waits for the system service's response. The response exchange field in the request block directs the response to the correct exchange.

The *default response exchange* is a special case of response exchange. This exchange is automatically used as the response exchange whenever a client process uses the request procedural interface. A run file is assigned a default response exchange when it is first loaded into memory. Each new process created in a program must be allocated a unique default response exchange.

Direct use of the default response exchange (that is, using it when you are not using the request procedural interface) is not recommended. The use of the default response exchange is limited to requests of a synchronous nature. That is, the client, after specifying the exchange in a Request, must wait for a response before specifying the exchange again in another Request.

A *service exchange* is an exchange that is assigned to a system service at system build or when the system service is dynamically installed. The system service waits for requests for its services at its service exchange.

Exchange Allocation

Exchanges are allocated in three ways:

- Exchanges for certain built-in system services are allocated at system build.
- Exchanges can be dynamically allocated and deallocated using the AllocExch and DeallocExch operations.
- A unique default response exchange must be allocated for each new process created in a program that will use the request procedural interface. A process can determine the identification of its own default response exchange using the QueryDefaultRespExch operation.

Upon termination, the exchanges allocated to the terminating program are deallocated.

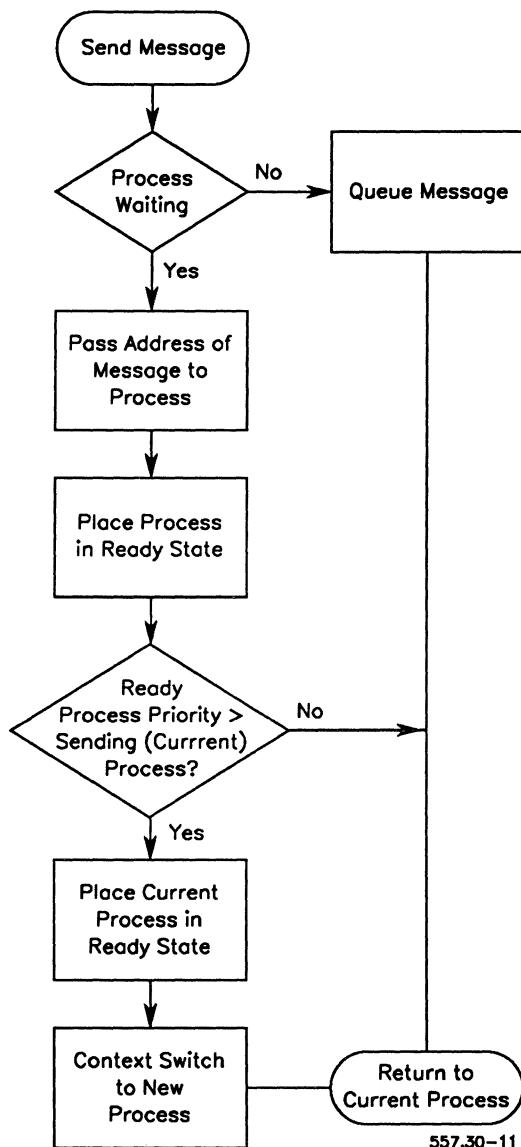
Sending a Message to an Exchange

When a process sends a message to an exchange, one of two actions results at the exchange (see Figure 30-11):

- If no processes are waiting, the message is queued.
- If one or more processes are waiting, the process that was queued first is given the message and is put in the ready state. If this ready process has a higher priority than the sending process, a context switch occurs, and the ready process becomes the running process. The sending process is placed in the ready state and loses control until it once again becomes the ready process with the highest priority. (The process states are described in “Process States” in the section entitled “Process Management.”)

After a message is queued at an exchange, the sending process must not modify it. A system service, for example, may have temporarily replaced the response exchange in a waiting client’s request block with its own service exchange to request a resource from another system service.

Figure 30-11. Sending a Message to an Exchange



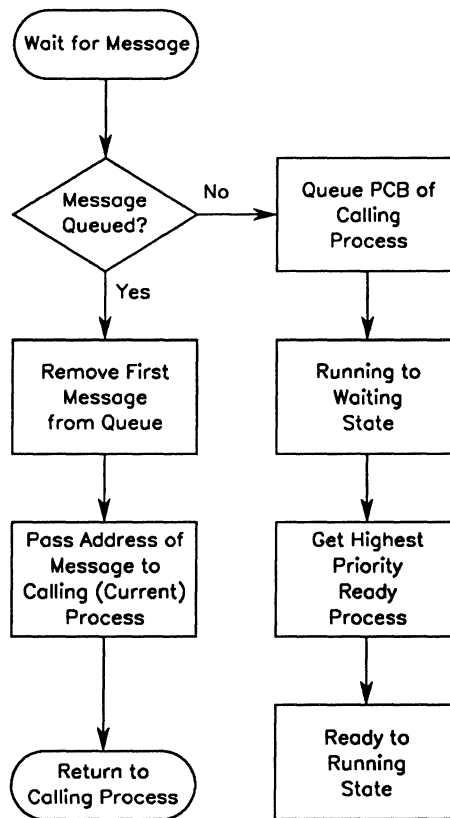
557.30-11

Waiting for a Message at an Exchange

When a process calls Wait and waits for a message at an exchange, one of two actions results at the exchange (see Figure 30-12):

- If no messages are queued, the calling process is placed in the waiting state, and its Process Control Block (PCB) is queued at the exchange until a message is sent.
- If one or more messages are queued, the message that was queued first is removed from the queue and its memory address is returned to the process, which then resumes execution.

Figure 30-12. Waiting for a Message at an Exchange

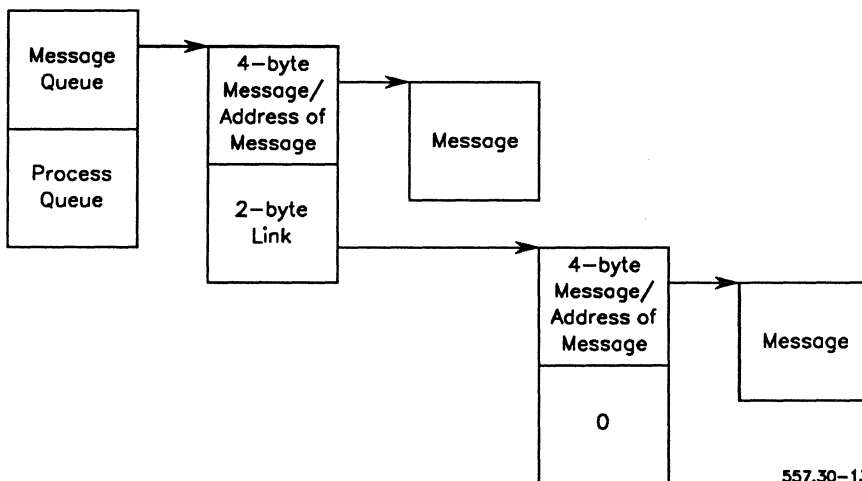


557.30-12

Exchange Queues

Either processes or messages, but not both, can be added to a queue at an exchange at any given time. Messages are queued using link blocks. A *link block* is a 6 byte structure containing the address of the message (or the message itself) in the first 4 bytes and the address of the next link block (if any) in the last 2 bytes. Figure 30-13 shows how messages are queued at an exchange.

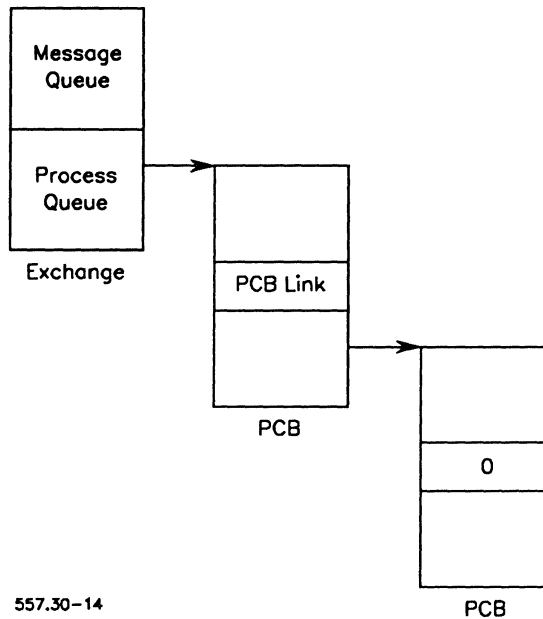
Figure 30-13. Messages Queued at an Exchange



Processes are queued at an exchange by linking through a field that is reserved for this purpose in each PCB.

Figure 30-14 shows how processes are queued at an exchange.

Figure 30-14. Processes Queued at an Exchange



The Message

A *message* conveys information and provides synchronization between processes. A 4 byte data item is communicated between processes. This data item is usually the memory address of a larger data structure, which is called the message. The interpretation of the 4 byte field is by agreement of the sending and receiving processes. Typically this field is the memory address of a request block.

The message can be in any part of memory that is under the control of the sending process. By convention, control of the memory that contains the message is passed along with the message.

A request block is a special type of message formatted according to specific conventions and used by all interprocess communications to the operating system. It is a data structure provided by the client, containing the specification and the parameters of the desired system service. A request block contains a request code field, a response exchange field, and several other fields; IPC is used most commonly with messages in this format.

This format is described in “Request Block Format,” below.

Request Block Format

The Request primitive initiates the request for a system service; the Respond primitive initiates the response. This structure provides

- Guaranteed matching of Requests and Responds
- Opportunity to redirect requests for system services to other system services
- Opportunity to redirect requests for system services to the server of a cluster configuration or over the network

The format of a request block is designed to pass information between a client and a system service. It provides for the transparent migration of system services between standalone, cluster, and network configurations.

Request blocks are self-describing and consist of the following parts:

- A standard header
- Control information specific to the request
- A routing code
- Descriptions of the request data items
- Descriptions of the response data items

Each of these components is described in the paragraphs that follow.

Standard Request Block Header

The format of the standard request block header is shown in Table 30-2.

Table 30-2. Format of a Request Block Header

Offset	Field	Size (bytes)
0	sCntInfo	1
1	rtCode	1
2	nReqPbCb	1
3	nRespPbCb	1
4	userNum	2
6	exchResp	2
8	ercRet	2
10	rqCode	2

sCntInfo

Is the number of bytes of control information. *Control information* is the data after the request block header and before the first request address/size (pb/cb) pair.

rtCode

Is a routing code placed in the request block by the operating system for routing requests. The default value of this field is 0.

nReqPbCb

Is the number of request address/size (pb/cb) pairs.

nRespPbCb

Is the number of response address/maximum size (pb/cbMax) pairs.

userNum

Is a 16 bit integer that uniquely identifies the client's partition and the resources with which it is associated.

Each application partition has a unique user number. The processes in an application partition share the same user number. A process can obtain its user number by means of the GetUserNumber operation.

If the userNum field contains 0, the operating system substitutes the user number of the client initiating the request.

exchResp

Is the response exchange of the client. A special exchange called the default response exchange is allocated when a run file is loaded into memory. It is used by the request procedural interface and should not be used explicitly. The AllocExch operation should be used to allocate exchanges.

ercRet

Is the status code (erc) returned by the system service.

rqCode

Is a request code, a 16 bit value that uniquely identifies the selected system service.

The request code is used both to route a request to the appropriate system service exchange and to specify to that service which of its functions is being requested.

Control Information

Control information is specific to each request. The field *sCntInfo* contains the number of bytes of control information transmitted from the client to the system service.

Routing Code

The routing code field *RtCode* consists of 1 byte that allows the kernel and agents to route a request from a program anywhere in the network, even if the request is undefined in the client's workstation operating system. The default value of this field is 0.

This field is important to you if you are defining requests for a system service. (See "Routing Requests" for more information on the field *RtCode*. Also see the section entitled "Inter-CPU Communication," for details on defining requests for system services to be run on shared resource processors, and the section entitled "System Services Management," for general information on defining requests for workstations or shared resource processors.)

Request Data Item

Each request data item descriptor consists of the following:

- The 4 byte memory address of the request data item
- The 2 byte size of the item

The total size (in bytes) of the request data item descriptors is 6 times *nReqPbCb*. Request data items are transmitted from client to system service. As an example, a request data item can be a name of a file to be opened.

Response Data Item

Each response data item descriptor consists of the following:

- The 4 byte memory address of the area into which the response data item is to be moved by the system service
- The 2 byte maximum allowable byte count of the response data item

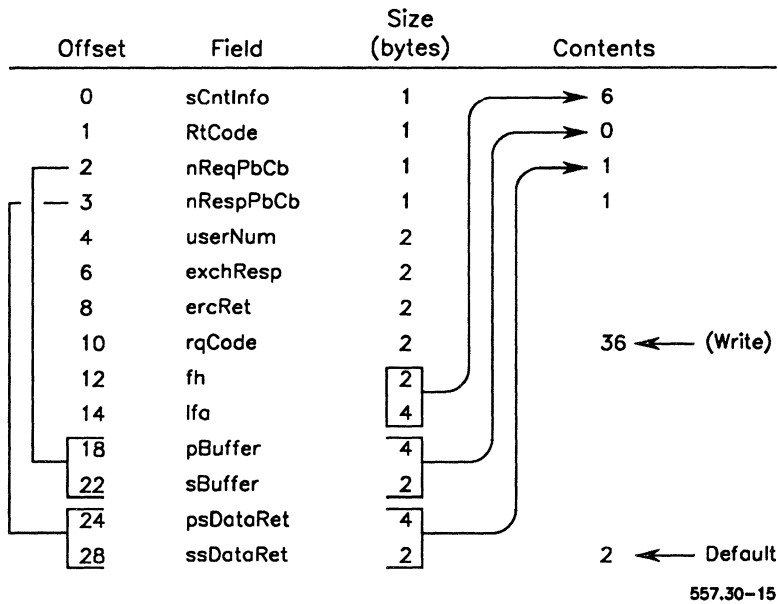
The total byte size of the response data item descriptors is 6 times *nRespPbCb*. Response data items are transmitted from system service to client.

9A response data item is information that the system service wants the client to know about, such as the file handle (*fh*) returned by the OpenFile operation or the number of bytes it wrote to the client's buffer in a Write operation.

Example Request Block

Figure 30-15 shows the request block for the Write operation.

Figure 30-15. Request Block for the Write Operation



Note that the request block header is the standard format described in “Standard Request Block Header.” The Contents column shows values for some of the request block fields, for example:

- The request code (*rqCode*) for Write is 36.
- The 6 bytes of control information (*sCntInfo*) consist of
 - The file handle (*fh*) returned from a previous OpenFile operation (2 bytes)
 - The logical file address (*lfa*) of the sector into which the data is to be written (4 bytes)
- The routing code (*RtCode*) field contains 0 until the request is issued. Information on resource handles and network routed specifications defaults to this field.
- A single request data item (*nReqPbCb*) is described by *pBuffer/sBuffer*.
 - *pBuffer* is the memory address of a buffer containing the data to be written.
 - *sBuffer* is the count of bytes to be written.
- A single response data item (*nRespPbCb*) is described by *psDataRet/ssDataRet*.
 - *psDataRet* is the memory address of a word into which the count of bytes successfully written is returned.
 - *ssDataRet* indicates the size of the word (2 bytes) into which the count of bytes written is returned. If the request procedural interface is used, it automatically supplies this value.

Accessing System Services

System services can be accessed

- Indirectly, by the request procedural interface
- Directly, by the kernel primitives, Request and Wait (or Check)

Using the Request Procedural Interface

Using the request procedural interface is convenient because it automatically constructs a request block and issues the Request and Wait primitives for you.

Except for the ReadAsync and WriteAsync procedures, the request block is constructed on the stack of the client process.

Most request procedural interfaces to system services do not provide any overlap between computation by the client process and execution of the system service. Because Read and Write are the system services for which the overlap of computation and execution of the system service is most ideal, however, the operating system provides the ReadAsync, CheckReadAsync, WriteAsync, and CheckWriteAsync operations.

These operations allow the client to initiate an I/O operation and then to compute and/or initiate other I/O operations before checking for the successful completion of the original one.

A special form of request procedural interface called the *alternate request procedural interface* provides a further convenience to clients that want to request a service on behalf of a different user number. The very nature of a system service may require that it issue the same request repeatedly for different user numbers.

The alternate request procedural interface requires only that you add the letters **Alt** and one extra parameter (*userNum*) to the parameters in the request statement. To write to a file, for example, you would write a statement of the form shown below:

```
erc=AltWrite(userNum, fh, pBuffer, sBuffer, lfa, psDataRet)
```

In the statement, *userNum* is the user number on behalf of which the request is being issued.

Using the Kernel Primitives Directly

Using the Request and Wait (or Check) primitives is more powerful than using the request procedural interface: it allows a greater degree of overlap between multiple I/O operations and computation by the client process.

For example, if you use the Check primitive instead of Wait, your program can continue executing some other function, such as updating the video to reflect current statistics. Execution is asynchronous with the return of the request.

To use the kernel primitives directly, your program must do the following:

- Call AllocExch to allocate a response exchange for the request block. Do not use the default response exchange.
- Build a request block in your program. Static information, such as the request block header, can be defined during program initialization. (See “Request Block Format.”) Parameters, such as buffers that change during program execution, must be updated each time the block is reused.
- Call the kernel primitives, Request and Wait (or Check).

Note: *Save the request block response exchange in a variable. Do not pass the response exchange in the request block as a parameter to a kernel primitive. Modification of an outstanding request block can result in conflict if, for example, the request block is redirected to a filter.*

If you need to synchronize program execution with the return of the request block information, your program can call Request and then issue a Wait for the response. Wait suspends process execution until the request block returns.

If your program does not depend on the information being returned immediately, it can issue `Check` periodically. `Check` tells whether a request block has returned without suspending program execution.

Your program may reach a point at which it must synchronize execution with the return of the request block. For example, your program may be performing a heavy computation, occasionally needing to write output to a disk file. When it is time to write, it can call `Wait`, specifying the response exchange in the request.

When the request block returns, your program can safely use it in another `Write` request. This may require adjusting the addresses and sizes of the request block buffers.

If more than one request block is outstanding, you must ensure that it is the correct one. To do this, your program can verify the request code or the address of the request block. (The request block address can be verified using the `FComparePointer` operation.)

Cluster/Network Communication

The operating system provides for cluster workstation as well as network configurations.

A *cluster configuration* consists of *cluster workstations* and a *server*. A cluster provides communication and resource sharing within a localized area, such as a building.

A *network configuration* consists of *nodes* connected by communications lines over long distances. A *node* is a junction in a network (such as a workstation or a processor board on a shared resource processor), where communication lines originate and/or terminate. Thus communication and resource sharing are provided over a wide area.

Cluster Configuration

The server of a cluster configuration can be a server workstation or a shared resource processor. The server provides resources, such as file system management and queue management, for all workstations in the cluster. In addition, it concurrently supports its own program processing as well as user-written, multiuser system services.

Essentially the same operating system executes in each cluster workstation as in a server workstation or in the combined processors of a shared resource processor. A cluster workstation can have its own local file system, or it can use the file management system of the server.

In the cluster configuration, IPC is extended to provide transparent access to system services that execute at the server. While some services (such as queue management, 3270 Terminal Emulator, and database management) migrate to the server, others (such as video management and keyboard management) remain at the cluster workstation.

Workstation Agent

In a cluster workstation, however, if the function is to be performed at the server, the request block is queued at the exchange of the workstation agent. The workstation agent converts interprocess requests to interstation messages for transmission to the server. The workstation agent is included at system build in a system image that is to be used on a cluster workstation.

Master Agent

The system image used at the server is built to include a corresponding service process: the master agent. The master agent reconverts the interstation message to an interprocess request. It queues the request at the exchange of the system service on the server that actually performs the intended function. Note that the operating system request routing table that translates request codes to service exchanges at the server is necessarily different from the table at the cluster workstation. When the system service at the server responds, the response is routed through the master agent, the high-speed data link, and the workstation agent before being queued at the client's response exchange in the cluster workstation that was specified in the request block.

The format of request blocks is designed to allow the workstation agents and master agents to convert between interprocess requests and interstation messages very efficiently and with no external information. Because request blocks are completely self-describing, the agents can transfer requests and responses between the server and cluster workstations without any knowledge of what function is requested or how it is to be performed.

Extending Resource Sharing

Network routing extends the operating system's resource sharing capability to permit sharing of system resources among nodes.

A system service, for example, can be installed at a server and accessed by remote nodes over the network or by workstations in a localized cluster.

Network routing makes use of two system services that issue and execute requests on behalf of clients and system services at local or remote nodes. These services are the Net agent and the Net server.

The Net agent receives requests destined for system services located at remote nodes and forwards these requests to the remote nodes.

The Net server responds to requests from remote Net agents. The Net server receives a request block from the Net agent, executes the request on behalf of the remote client, and returns the response to the originating Net agent.

(For more information on the network environment, see your CT-Net or B-Net manual.)

Routing by Handle

A request to use a resource can be routed by handle. A *resource handle* is a 16 bit identifier assigned to the resource by a system service. The handle is used by the system service to reference the resource in subsequent requests the client makes to use the resource again.

For example, when a client calls `OpenFile`, the file system returns a file handle for the file opened. In requests to read from or write to the file, the client passes the handle back to the file system.

Resource handles (except file handles) uniquely identify an open resource. File handles are unique for a given user number.

Certain pre-established fields of the request block contain the resource handle information. (See “Request Block Format.”) A system service returns a handle in the first response pb/cb pair. When a client issues a request by handle, the handle is the first word of the request block control information.

The bits in a resource handle have different meanings depending upon the following factors:

- How the request is defined. On a shared resource processor, for example, a request is routed according to the routing type defined for it. (Shared resource processor routing types are discussed in “Shared Resource Processor Routing” and in “Shared Resource Processor Routing Types” in the section entitled “Inter-CPU Communication.”)
- Whether the request to use the resource is being viewed by the system service or the client. Handles routed over the network are seen differently by the client and system service.
- The version of the operating system upon which the software is being run.

Because the responsibility for building and issuing resource handles lies with system services, it is important that you understand how handles differ if you are writing a system service program. (For details on using handles on a shared resource processor, see “Writing System Services for the XE-530” in the *CTOS Programming Guide*.)

The client program does not need to know the exact details of each bit in the handle. To access a given resource previously opened, all the client is required to do is to return the correct handle in the appropriate request block field. If, however, you are debugging a program that calls a system service, you can expect the bit fields in all handles returned by system services to have the same general meanings shown in Table 30-3.

Table 30-3. Interpretation of Resource Handle Bits

Bits	Meaning
0 to 12	These bits identify the resource opened. In most cases these bits can mean anything the system service wants to use them for. With some shared resource processor routing types, some of the bits have special meanings. (For details on handle building, the system service writer should see the <i>CTOS Programming Guide</i> .)
13 and 14	These bits indicate where the system service is installed. If either of these bits is set (to 1), this means the system service is at a server. Otherwise, the system service is at the local workstation.
15	This bit identifies a network handle. This bit is set by the Net agent at the local server when a handle is built and returned by a system service at a remote network node. Any outbound request containing a network handle is routed to the Net agent at the local server for forwarding to the correct node.

The Net agent uniquely maps resource handles arriving from remote nodes to local handles. Figures 30-16, 30-17, and 30-18 (below) illustrate how handles are built and used to route requests over the network. The figures show a client executing on a local cluster workstation and a system service executing at a server on a remote node. The details of resource handle management shown in each figure are described in the following paragraphs.

Note: *The details of how a request is routed to a system service are not fully disclosed in the examples illustrating the resource handle concept below. Resource handles tell part of the story in cases involving handles and network routing. See the section entitled “System Services Management,” for additional information on routing from the viewpoint of the system service.*

Figure 30-16 shows the client requesting a resource from the system service at a remote network node. In this example, the resource is a file that the client wants the service to open. Each labeled part of the figure is described below. In each case, the value of the handle is shown.

Label	Description
A	The client calls <code>OpenFile</code> . The request is routed to the Net agent at the server.
B	The Net agent forwards the request over the network to the system service.
C	The system service builds file handle number 2304h.

Figure 30-16. Requesting to Open a File

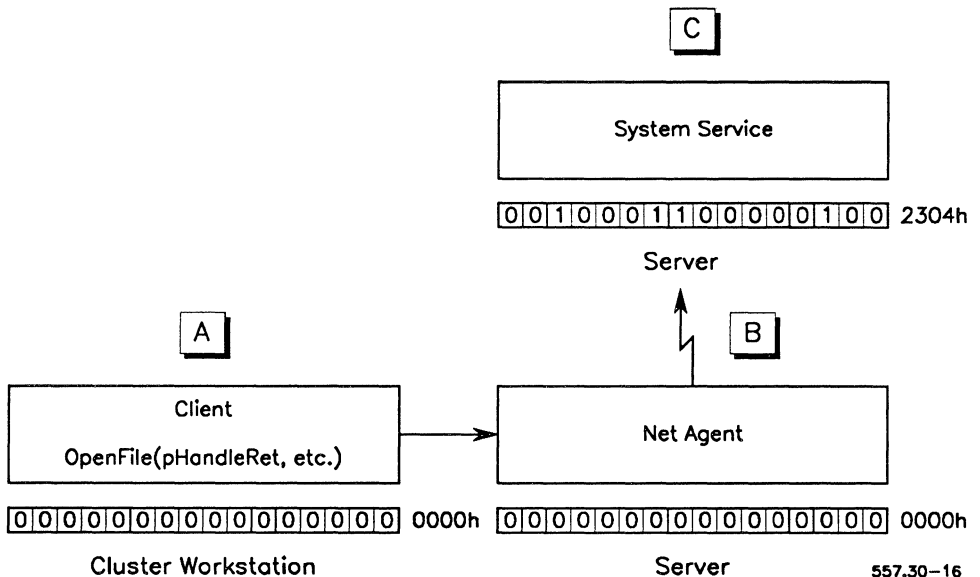


Figure 30-17 shows the response to the open client request illustrated in Figure 30-16. Each labeled part of the figure is described below. In each case, the value of the handle is shown.

Label	Description
C	The system service has built file handle number 2304h. Bit 13 in the handle is set (is 1) because the system service is executing at a server. (See Table 30-3.)
B	The Net agent creates local handle number 0A001h in its handle mapping table and associates this handle with handle 2304h and the system service.
A	The Net agent returns handle 0A001h to the client. Whenever a resource handle is returned by a remote system service in the network, the Net agent always sets bit 15 in the corresponding local handle.

Figure 30-17. Responding With a File Handle

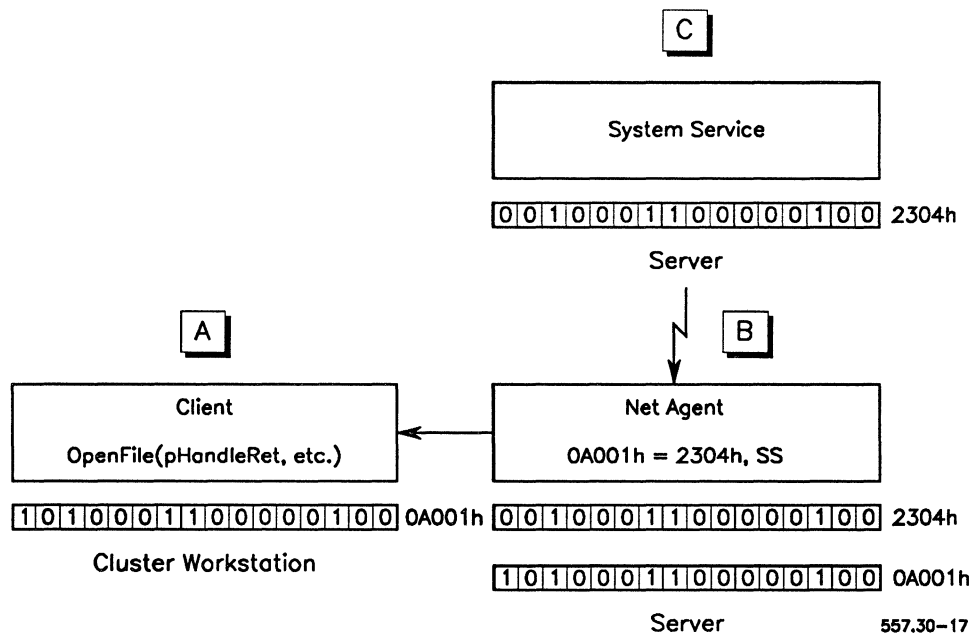
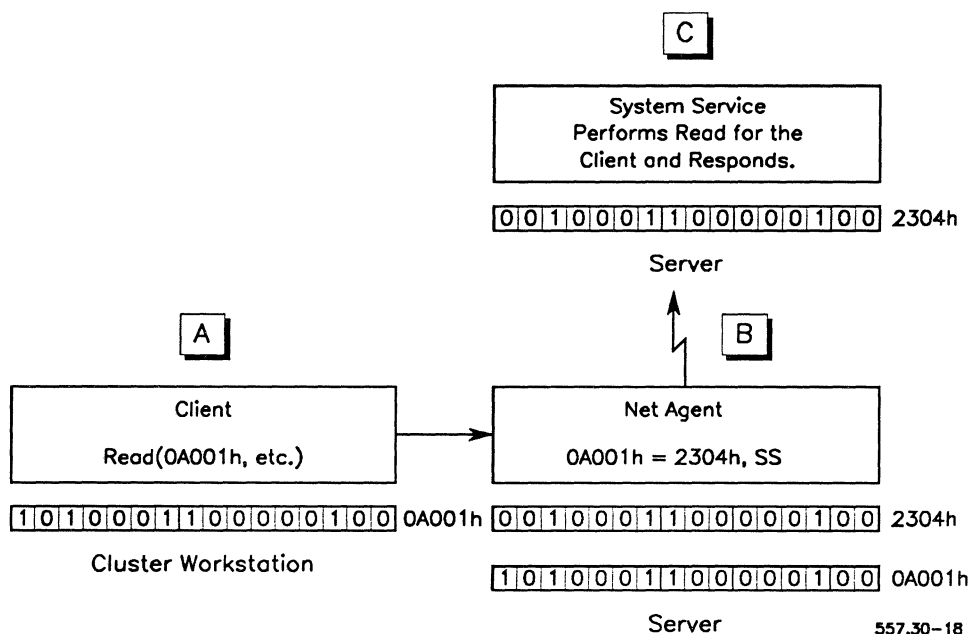


Figure 30-18 shows the client using the handle returned (in Figure 30-17). Each labeled part of the figure is described below. In each case, the value of the handle is shown.

Label	Description
A	The client provides handle 0A001h in the request to read the open file. Because bit 13 of the handle is set (is 1), the request is sent to the server. (See Table 30-3.) Furthermore, because bit 15 is set, the request is directed to the Net agent.
B	The Net agent examines its handle mapping table to see which system service and associated handle correspond to handle 0A001h. The table associates handle 2304h with the system service that built it (and the remote node where the service is installed). The Agent forwards the Read request with handle 2304h to this service.
C	The system service performs the read for the client.

Figure 30-18. Using the File Handle



The response would then be routed back to the client through the local server where service handle 2304h is again associated with client handle 0A001h.

Not mentioned in the preceding discussion are the workstation agent, master agent, and the Net server. (See "Cluster/Network Communication.") The workstation agent and master agent direct requests back and forth over the RS-485 or RS-422 cluster lines between the server and local workstations connected to it.

The Net server actually plays a very minor role at the receiving end of a request routed over the network and does nothing to influence the contents of the resource handle. In Figure 30-18, for example, the Net server (at the remote server) simply receives the request with handle 2304h from the Net agent (at the local server) and forwards the request to the associated system service. When the system service responds, the Net server directs the response back to the local Net agent.

In a complex network with many installed system services, it is possible and valid for system services from different nodes to build handles with the same values. In the Figures 30-16 through 30-18, for example, a system service at a different remote node also could build handle 2304h. To identify the source of each handle, the Net agent associates the handle built by each system service with a different local handle. The handle table used by the Net agent might contain several entries like the ones shown below, for example:

Client Handle	System Service Handle/ Associated System Service
0A001h	2304h, SS B
0A002h	2304h, SS C
0A003h	0001h, SS B
0A004h	1234h, SS B

Note that the client handle shows the top bit set (by the Net agent). A system service never builds or sees a handle with this bit set.

Routing by Specification

Requests can be routed by specification. Specifications are described by address/size (pb/cb) pairs in the request block. (See “Request Block Format.”)

Rules for Routing by Specification

A request routed by specification must adhere to the following rules:

- Node names are from 1 to 12 characters long and can be any combination of alphanumeric characters. Each node must be given a unique name and address.

Note: *In this section, the terms master and server are equivalent.*

Two node names are reserved:

Name	Meaning
local	Is ignored. Other routing information is used.
master	Route this request to the server.

- A request can have a maximum of two specifications. The first specification must be in the first request pb/cb pair; the second (if any), in the third pb/cb pair.
- If a specification has a password associated with it, the password must be specified by the pb/cb pair immediately following the specification. A second instance of the specification must also have the password.

Expanding Specifications

Any incomplete specifications are expanded before they are sent to the server. (The server does not have a copy of the user control block and therefore cannot expand the specifications itself.)

Expanding a specification involves adding default path information from the User Control Block. The information that must be added depends on the type of the specification.

Specifications are expanded as shown in Table 30-4.

Table 30-4. Specification Expansion

Specification Type	Method of Expansion
FileSpec	Expands everything to the left of the file name, that is, the default file name prefix, the default directory, the default volume, and the default node, for example: <i>{Node}[VolName]<DirName>FileName</i>
DevSpec	Expands everything to the left of the volume name, that is, the default node, for example: <i>{Node}[VolName]</i>
DirSpec	Expands everything to the left of the directory name, that is, the default volume and the default node, for example: <i>{Node}[VolName]<DirName></i>
FileSpec2	The same as FileSpec, but the request contains two specifications to expand.
FileSpecP2S2	The same as FileSpec, but the specification occurs in the third request pb/cb pair, instead of the first.

Routing Code

The routing code (*rtCode*) field is a 1 byte field of the request block used by the kernel and agents to route requests. It determines

- Whether the request is to be routed by specification or by handle
- For requests routed by specification, the location of the specification in the request block
- For requests routed by specification, the method of expansion

The request is defined with the *rtCode* values as described in Table 30-5 and 30-6.

Table 30-5 shows the values for routing the request. Table 30-6 shows the values for specification expansion.

Table 30-5. RtCode Values for Request Routing

Value	Token	Description
80h	RW	This request is a read or write and may have to be broken up into small requests. (This only applies to read or write requests.)
40h	OpenFh	This request opens a resource. The first response pb/cb pair of this request returns a handle that is used later by other requests to refer to the resource.
8h	rFh	This request is routed by handle. The handle was returned by a request defined as OpenFh.
6h	CloseFh	This request closes a resource that was opened by a request defined as OpenFh.

Table 30-6. RTCode Values for Specification Expansion

Value	Token	Description
0h		No specification routing.
1h	DevSpec	Route by device specification.
2h	DirSpec	Route by directory specification.
3h	FileSpec	Route by file specification.
4h	FileSpec2	Route by file specification. (The request contains two of them.)
5h	FileSpecP2S2	Route by file specification in P2/S2.
10h	SpecPW	In addition to selecting one of the above values, SpecPW can be used for passwords. All specification pb/cb pairs are followed by password pb/cb pairs. If SpecPW is used and there is no specification to expand (rSpec = 0 or rSpec > 5), the first pb/cb pair is a password to expand. (ChangeOpenMode is an example).

Routing Requests

A client request can be routed from anywhere in a cluster or network, even if the request is undefined in the client process's workstation operating system. In a standalone workstation, the request block is queued at the exchange of the system service that actually performs the desired function.

When a client calls `Request`, any number of events can happen to route the request to the destination where the appropriate service exchange is located. In the description of the `Request` primitive (in "Request and Respond"), you will recall that the request code serves as an index to routing information contained in the operating system request routing table. The kernel consults this table to determine the following:

- Whether the request involves network routing (the request block contains either a handle or specification).
- Whether the request is to be routed on a shared resource processor. This information is derived from the request definition. (More is said about defining requests in later sections.)
- The service exchange to which the request is to be routed. The service exchange is established at the time the system service installs and calls `ServeRq` to serve the request. (This is explained in the guidelines for writing system services in the section entitled "System Services Management.")

Based on this information, the kernel makes the routing decisions illustrated in the flow charts shown below in Figure 30-19, Figure 30-20, and Figure 30-21. In each figure the circled numbers 1, 2, and 3 have the following meanings:

Number	Meaning
1	Go to Figure 30-19.
2	Go to Figure 30-20.
3	Go to Figure 30-21.

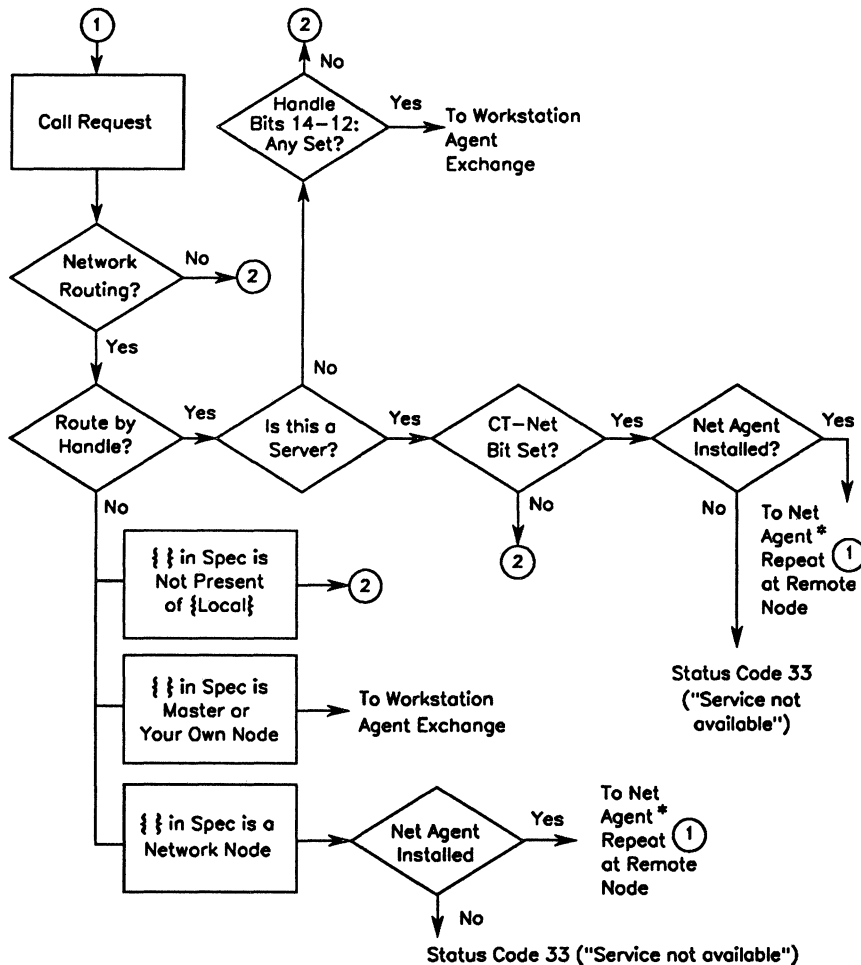
For example, an arrow pointing to number 2 in a figure means request routing continues as depicted in the flow chart in Figure 30-20.

Comments on the figures are the subject of the following paragraphs.

Network Routing

In Figure 30-19, the first figure in this three-part series on request routing, the kernel determines if the request involves network routing. If so, the kernel further determines the type of network routing (that is, whether routing is by handle or specification).

Figure 30-19. Network Routing



* SRP Only: Net Agent is always on the master processor.

557.30-19

See the decision block “Route by Handle?” in Figure 30-19. The Yes branch means routing by handle; otherwise the request is routed by specification.

For requests routed by specification, the kernel examines the value of the node specification and routes the request accordingly. In cases of routing by handle, the kernel makes different decisions depending on whether the request was issued at a server or a cluster workstation. If the request was issued at a cluster workstation, the kernel examines bits 12 through 14 of the handle. Requests with a nonzero value for any of these bits are directed to the workstation agent for transmission to the server. If, however, the request was issued at the server, the kernel checks to see if the network routing bit is set. If so, the request is sent to the Net agent for forwarding to the appropriate node. (For details on handles, see “Routing by Handle.”)

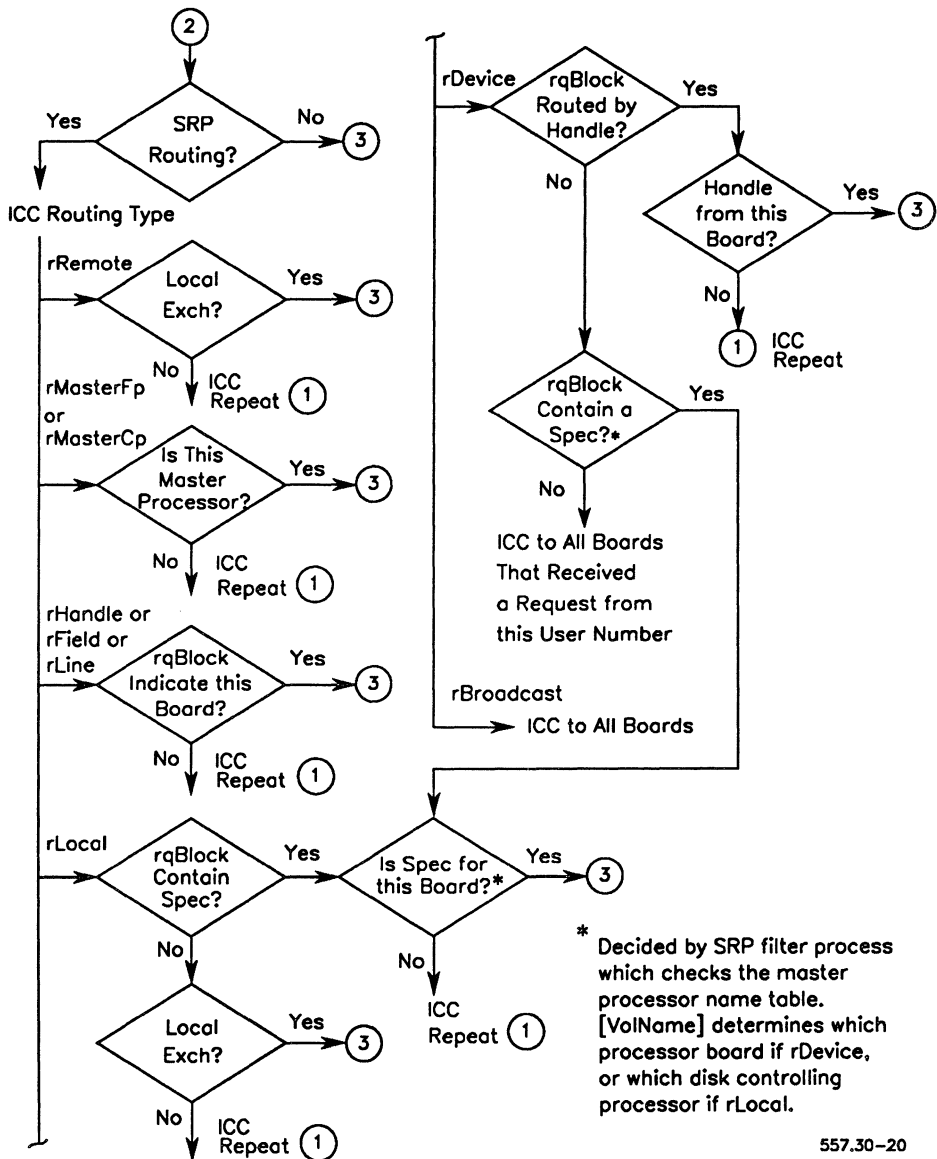
Before routing any request over the network, the kernel ensures that a Net agent is installed. If an agent is not installed, routing terminates with status code 33 (“Service not available”). Requests not destined to a different network node are routed as described in “Shared Resource Processor Routing,” below.

Shared Resource Processor Routing

In Figure 30-20, the second figure in this three-part series on routing, the first decision “SRP Routing?” is whether the request involves SRP routing. Requests routed on a shared resource processor are defined by one of nine shared resource processor routing types (described in detail in “Shared Resource Processor Routing Types” in the section entitled “Inter-CPU Communication.”). Otherwise they are routed by exchange as described in “Exchange Routing.”

Based on the shared resource processor routing type, the kernel makes further decisions. In each case, however, there is a general pattern: the request is either directed to a service exchange, or it is routed by means of inter-CPU communication (ICC) to another board where the process of calling Request is repeated.

Figure 30-20. Shared Resource Processor Routing



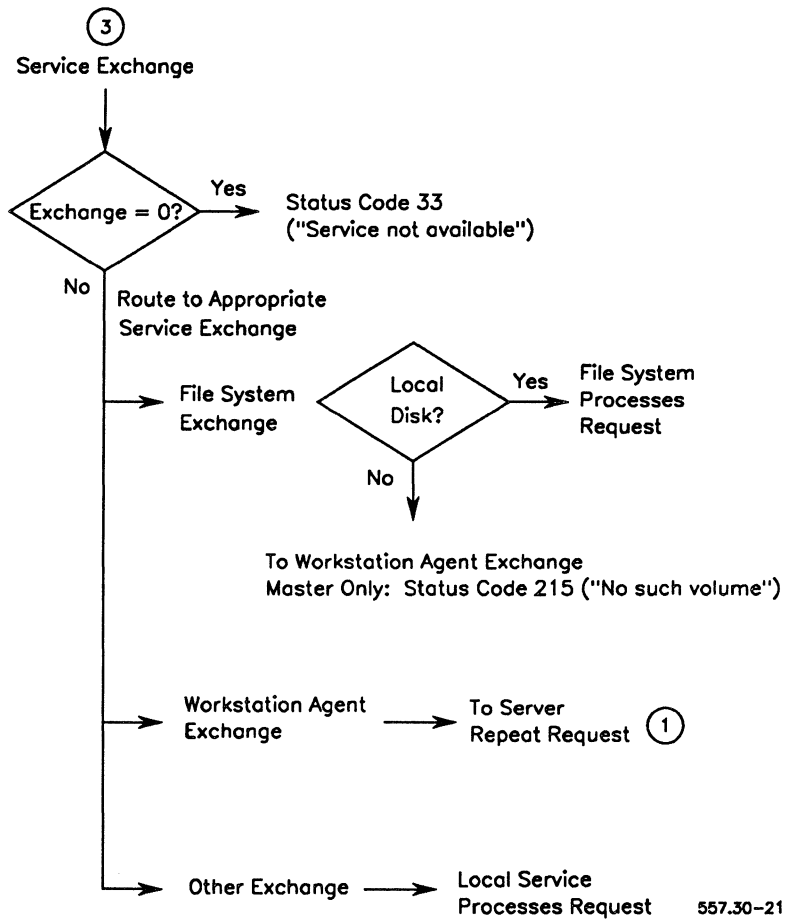
557.30-20

Figure 30-20 shows the shared resource processor filter process (rather than the kernel) making the decision for routing types *rLocal* and *rDevice*. (Perhaps more significant is the fact that network routing, combined with the shared resource processor routing type, determines the final destination of a request defined *rLocal* or *rRemote*. For details, see the descriptions of these routing types in “Shared Resource Processor Routing Types” in the section entitled “Inter-CPU Communication.”)

Exchange Routing

Finally, Figure 30-21, the third figure in this three-part routing series, shows the kernel decisions based on the service exchange. A service exchange value of 0 means that no system service called *ServeRq* to serve the request. In this case, the kernel returns status code 33 (“Service not available”) and terminates routing. Otherwise, the kernel compares the exchange value in the request to all the service exchanges and routes the request to the appropriate exchange. The figure shows the file system and workstation agent exchanges simply to illustrate how the request might be routed back to the server again where the process of calling *Request* is repeated.

Figure 30-21. Exchange Routing



557.30-21

Filter Process

A *filter process* is a system service that is interposed between a client and a system service process so that they appear to be communicating directly with each other. The filter does this by substituting its exchange for that of the original system service in the operating system request routing table.

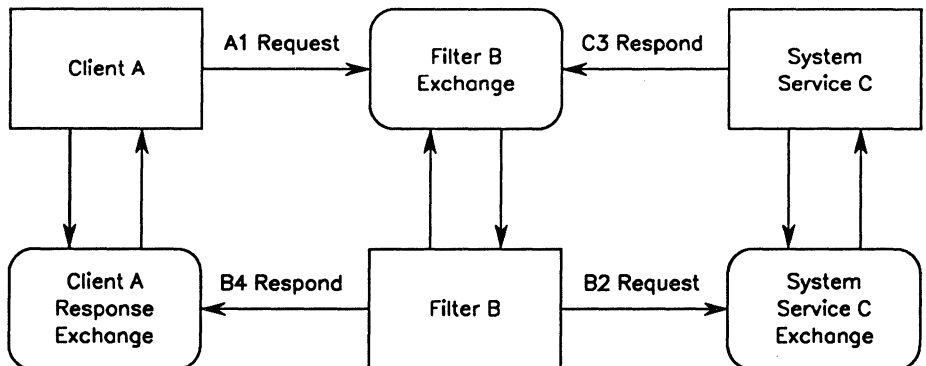
Filters can be used in many ways. A filter, for example, might be used between the file management system and its client process to perform special password validation on all or some requests. Filters are commonly used by the keyboard service to filter keystrokes for various accounting purposes.

The interaction of a filter process with a client and system service process is shown in Figure 30-22.

Workstation agents and Net agents act as filters in directing IPC messages to other destinations for further IPC processing. Configurations involving network routing require that a filter intercept messages branching to local services as well as those that are routed over the network.

(For details on filters, see “Filters” in the section entitled “System Services Management.”)

Figure 30-22. Filter Process Interaction



557.30-22

Interprocess Communication Summary

Figure 30-23 summarizes interprocess communication concepts presented in this section.

Figure 30-23. Interprocess Communication Summary

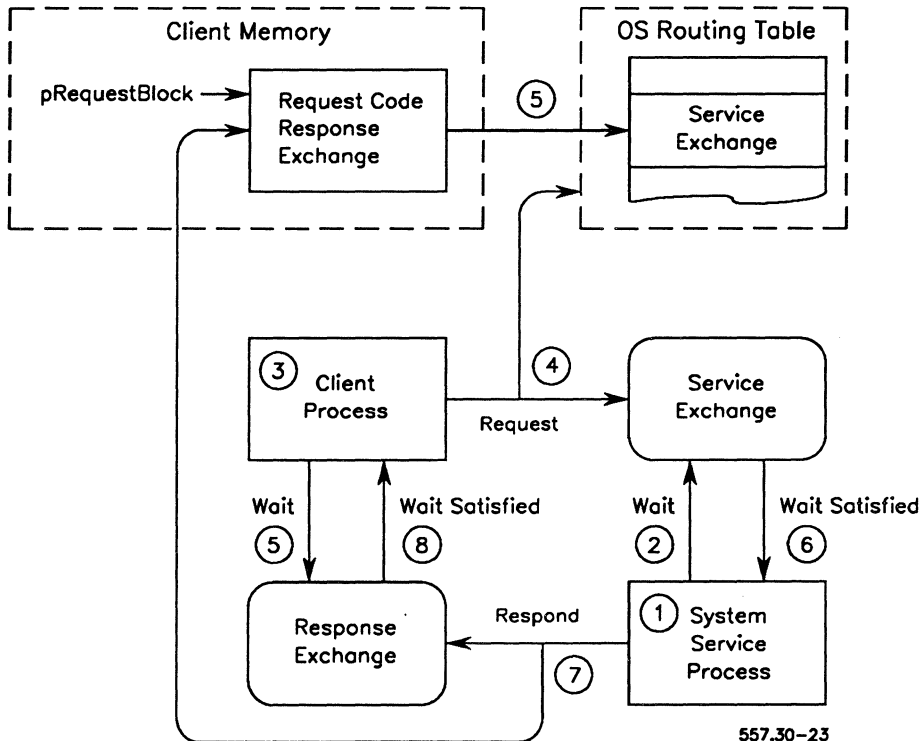


Figure 30-23 depicts the following event sequence:

1. The system service calls `ServeRq` to serve the request code(s) at its exchange. (`ServeRq` is discussed in “Guidelines for Writing a System Service” in the section entitled “System Services Management.”) This causes the kernel to place the service exchange in the operating system request code routing table at the offset of the request code.
2. The system service process waits at its service exchange. (The kernel takes the system service process off the run queue and places it in the wait state.)
3. The client process builds a request block in its memory. (Note that the request procedural interface will automatically do this step and steps 4 and 5 for the client.)
4. The client calls `Request`. (The kernel looks up the service exchange in the operating system request routing table and queues the request block address on the service exchange message queue. The request can be routed over various paths as described in “Routing Requests.”)
5. The client issues a `Wait`. (The kernel takes the client process off the run queue and queues the client at its response exchange. The response exchange is the default response exchange if the request procedural interface is used.)
6. The kernel removes the request block address from the service exchange message queue and passes it to the system service process. The system service process is placed in the ready state.
7. The system service performs its function and calls `Respond`. The kernel looks up the client’s response exchange in the request block and routes the request back to the client.
8. The client process is given the request block and is placed in the ready state. If it is the highest priority process, it is given control of the processor, and it continues execution.

Interprocess Communication Operations

The IPC operations are described below. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Request

Requests a system service by sending a request block to the exchange of the system service process.

QueryDefaultRespExch

Allows a process to determine the identification of its own default response exchange.

Wait

Removes the message (if any) that was queued first from the queue at the specified exchange. Wait causes the calling process to be placed into the waiting state if no messages are queued.

WaitLong

Similar to Wait but is used if the process waiting is expected to be waiting for a long time (more than 30 seconds).

AllocExch

Allocates an exchange.

Respond

Notifies a client process that the requested system service was performed by sending the request block of the client process back to the response exchange specified in the request block.

Check

Removes the first message queued (if any) at the specified exchange. Check returns the status code 14 ("No message available") if no messages are queued.

Send

Sends the specified message to the specified exchange.

RequestDirect

Sends a request block to an explicitly specified system service exchange. Sending the request block is done independently of the default routing implied by the request code in the request block.

ForwardRequest

Used by a filter process to forward a request block to another system service for further processing. It does not require a matching Respond.

PSend

Functions identically to the Send primitive but is used instead of Send for interrupt handling.

DeallocExch

Deallocates an exchange.

InstallNet

Passes network routing installation parameters to the operating system(s) of a server.

Section 31

Semaphores

The operating system provides a set of semaphore operations to create and operate on semaphores. This section describes semaphores and provides guidelines on how they can be used.

What is a Semaphore?

A semaphore can be thought of as a switch with an on and an off state. An application can use a semaphore in two basic ways:

- To ensure exclusive access to a resource by a single process (mutual exclusion)
- To signal processes and synchronize their execution

To turn the semaphore on, an application *sets* or *locks* it. An application *clears* the semaphore to turn it off. The semaphore state is contained in a single bit called the *semaphore lock bit*, which is set or cleared when an application calls a semaphore operation.

A separate set of semaphore operations is provided to operate on semaphores used for mutual exclusion and for process synchronization.

Internally a semaphore contains information allowing the system to manage its use. For example, this information indicates

- The semaphore state
- Which process is currently using the semaphore
- What processes are waiting to use it

A semaphore is specified by a semaphore handle.

Who Uses Semaphores?

Semaphores are supported on CTOS primarily to facilitate porting Presentation Manager and other applications from operating systems such as OS/2 that are not message based. Applications written to run on such systems use semaphores extensively.

Note: *If you are porting applications, there are a few differences in the implementation of CTOS semaphores you should be aware of. (For details, see the CTOS User Interface Programming Handbook.)*

If you are just beginning to write applications to be run on CTOS, it is recommended that you use interprocess communication (IPC) kernel primitives rather than the semaphore operations. Ensuring exclusive access to shared resources and synchronizing processes are features inherently built into distributed, message-based operating systems like CTOS. In most cases the same kind of functionality that you get with a semaphore is available by using kernel primitives such as Request, Send, and Wait. This section illustrates how these kernel primitives compare to the semaphore operations. (For details, see “Mutual Exclusion on CTOS” and “CTOS Solutions to the Producer/Consumer Model.”)

The only cases where you would actually need to use semaphores in CTOS applications is if you are writing a dynamic link library (DLL) or a special type of system service called a *system-common service*. Neither DLLs nor system-common services enforce the serial execution of processes. As such they require semaphores to ensure that resources are used by one process at a time. In this situation, the semaphore operations provide a standardized way to operate on semaphores that you can take advantage of rather than writing semaphore management routines on your own. (For further information on system-common services, see the section entitled “System-Common Services Management.” For a comparison of DLLs to system-common services, see “General Guidelines for Writing DLLs,” in the section entitled “Dynamic Link Libraries.”)

Semaphore Terminology

Table 31-1 presents you with the terms you need to become familiar with as you read this section.

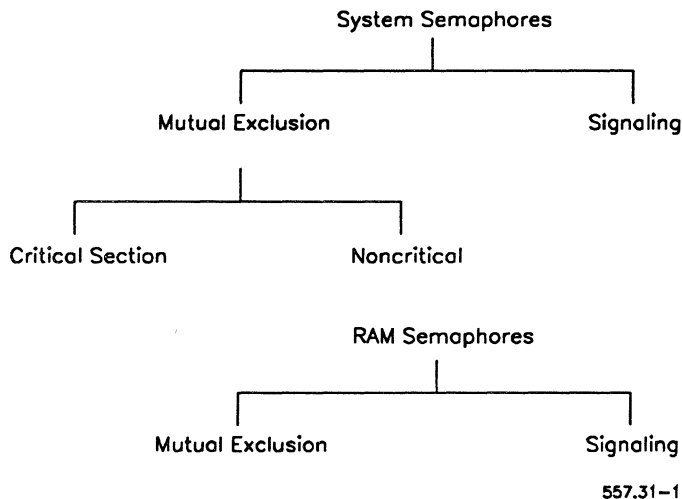
Table 31-1. Semaphore Terms

Term	Definition
Atomic	Referring to an operation that, once started, completes without interruption.
Clear	Turn the semaphore off. Clearing a semaphore resets the semaphore lock bit to 0.
Critical section	A portion of code in which a process accesses shared, modifiable data.
Lock or set	Turn the semaphore on. Locking or setting a semaphore sets the semaphore lock bit to 1.
Mutual exclusion	Allowing only one process at a time access to a shared, modifiable resource.
Noncritical semaphore	A mutual exclusion semaphore that may be held for a long period of time and is locked with the SemLock operation.
Nonterminatable	A process owning this type of semaphore cannot be terminated until it clears the semaphore.
RAM semaphore	A local semaphore. The handle of a RAM semaphore points to a semaphore variable created in the data space of the caller.
Semaphore handle	The memory address of the semaphore variable.
Semaphore lock bit	The low bit of the semaphore variable low-order word.
Semaphore owner	The process locking the mutual exclusion semaphore. A semaphore remains locked until the semaphore owner clears the lock (or until the system cleans up the semaphore owner's resources at termination).
Semaphore variable	A double word pointed to by the semaphore handle. The low bit of the low-order word is the semaphore lock bit. For RAM semaphores, bit 8 of the low-order word indicates if a process is waiting for the semaphore.
Signaling semaphore	A semaphore used to announce the completion of a single event to one or more waiting processes.
System semaphore	A semaphore opened using the SemOpen operation. A system semaphore variable is maintained in internal system data structures that contain additional status information.

Semaphore Types

Semaphores can be any of the types shown in Figure 31-1.

Figure 31-1. Semaphore Types



System and RAM Semaphores

There are two basic types of semaphores: system and RAM. Each type can be used for mutual exclusion or for process signaling. System semaphores are maintained in internal data structures in system memory. RAM semaphores are created locally in the caller's data space.

Caution

Although CTOS supports RAM semaphores, they should only be used by applications ported to CTOS that already use them. New applications needing semaphores should use the operations for creating system semaphores instead.

If an application must use semaphores at all, it should use system semaphores. They are faster, safer, and provide more flexibility in the ways they can be used. As such, this section focuses primarily on system semaphores. The subject of RAM semaphores is taken up separately. (See “Using RAM Semaphores.”)

All system semaphores are opened with the SemOpen operation. SemOpen returns a semaphore handle that can be used in subsequent mutual exclusion or signaling semaphore operations. The SemClose operation is used to close the semaphore after a process is finished using it. (See “Opening a System Semaphore,” for details.)

Mutual Exclusion Semaphores

Mutual exclusion semaphores are used when only one process at a time can be allowed to access modifiable data or shared resources. That process must exclude all other processes from simultaneously executing the *critical section* of code that accesses the data or resource. For example, to add a new entry to a queue, it might also be necessary to update four words in memory. While the queue entry is being added, it is important that no other process access the same memory locations.

When a process wants to enter the critical section, it requests *ownership* of the semaphore. The operation locking the mutual exclusion semaphore checks the status of the semaphore *lock bit* before setting it in a single, *atomic* test-and-set operation. If the bit is already locked by another process, that process owns the semaphore until it clears the lock. In such a case, the requesting process may be required to wait for the semaphore. This ensures exclusive access to a shared resource such as a memory location or a device by the process currently owning the semaphore.

Critical Section Semaphores

Critical section semaphores are a subset of mutual exclusion semaphores. They are implemented in such a way as to economize on processor time. Critical section semaphores typically are used by the operating system and system services to protect the contents of shared system data structures they are updating.

Noncritical Semaphores

For convenience, the text refers to mutual exclusion semaphores that may be held for a long period of time as *noncritical semaphores*. Noncritical semaphores typically are used to control access to a shared resource such as a printing device where use of processor time is not a major concern.

Signaling Semaphores

A signaling semaphore is used when one or more processes need to wait for a single event to occur. Signaling semaphores have no concept of ownership. They're either set or cleared by signaling operations. When a process wants to wait for the event, it uses a separate Wait operation to do so.

The signaling function is separate entirely from mutual exclusion. As such, there is a special set of signaling operations designed just for this purpose. (See "Signaling and Synchronizing Processes.")

Note: *Signaling and mutual exclusion operations must not be used interchangeably.*

Opening a System Semaphore

An application opens a system semaphore using the SemOpen operation and closes it with the SemClose operation. Usually an application declares all the system semaphores it is going to open during initialization.

In the call to SemOpen, an application can define the system semaphore it wants to open to have additional characteristics. The *options* parameter allows the caller to specify whether or not

- The mutual exclusion semaphore is a critical section semaphore.
- A process locking a mutual exclusion semaphore can be terminated before it releases the lock.
- There is a restriction on which processes may actually obtain the semaphore.

The number of system semaphores that can be opened is a configurable parameter. (For details, see "Configurable Parameters" in Section 16 of the *CTOS System Administration Guide*.)

Providing Exclusive Access to a Resource

There are two types of mutual exclusion semaphores: noncritical and critical section. Each type has its own set of semaphore operations.

Noncritical Semaphores

If processor time is not a critical issue, your application can use the `SemLock` and `SemClear` operations to lock and clear a noncritical semaphore. Noncritical semaphores are used in situations where it is acceptable for the process to wait at an exchange, for example, to write data to a printing device that might be in use by another process. Sometimes, a waiting process can be bumped by another process of a higher priority before it actually gets the semaphore.

With this type of semaphore, there is a potential for much more processor time to be spent juggling processes onto and off the run queue than there is with critical section semaphores. (See “Critical Section Semaphores.”)

Locking a Noncritical Semaphore

`SemLock` normally is a very fast operation. It examines the semaphore lock bit. If it finds that the bit is clear, it sets it immediately in one *atomic* test-and-set operation. Otherwise `SemLock` causes the caller to wait at its response exchange until the semaphore clears before it can lock the semaphore for the caller. `SemLock` has a *timeout* parameter that allows the caller to control the semaphore waiting semantics.

A mutual exclusion semaphore can be locked again by a process that already owns the lock, as can occur when `SemLock` is called repeatedly in a nested procedure. If the mutual exclusion semaphore is a system semaphore the caller already owns, `SemLock` increments a use count, and the caller continues processing as if it just acquired the lock.

Clearing a Noncritical Semaphore

`SemClear` is paired with the `SemLock` operation to clear the noncritical semaphore specified by the semaphore handle. Like `SemLock`, `SemClear` is very fast for the most common case (that is, when no process is waiting for the semaphore). If, however, a process is waiting, a CTOS interprocess communication (IPC) kernel primitive is used to wake up the processes waiting for the semaphore. This may cause a process switch (and cost machine cycles) if a process waiting for the semaphore has a higher priority than the process calling `SemClear`.

If the semaphore is locked repeatedly by the same process, the call to `SemClear` merely decrements the use count associated with it. Only when the count reaches 0 does `SemClear` actually clear the lock.

Critical Section Semaphores

If processor time is an issue your application can use a critical section semaphore. It is sometimes necessary for a process to execute a critical section of code very quickly so other processes needing to use the resource do not have to wait too long.

A critical section semaphore locks the critical section before a process starts executing the code it contains. When execution is complete, the semaphore is cleared. Although the process can be interrupted, all other processes wanting to execute the same critical section are blocked until the first process is done.

Critical sections also can be implemented by disabling and enabling interrupts. This was the only way to implement critical sections before critical section semaphores were supported. Using the disable/enable method, interrupts are disabled before a process starts executing a critical section. Interrupts are not enabled again until the process completes the section.

This method is effective when critical sections are very short but the longer interrupts are kept disabled, the greater the chances are for interrupt latency problems to occur. A communications interrupt handler, for example, has very little patience. It cannot be interrupted for very long before it starts to lose bytes of data.

Using critical section semaphores avoids interrupt latency problems and is the preferred method of implementing critical sections. CTOS programmers are encouraged to use it.

Using the Critical Section Operations

The `SemLockCritical` and `SemClearCritical` operations are used to lock and clear a critical section semaphore. Although these operations are seemingly similar to the noncritical semaphore operations `SemLock` and `SemClear`, they are implemented differently to economize on processor time.

The main difference is that `SemLockCritical` and `SemClearCritical` do not cause a process to wait at an exchange for the semaphore to be cleared. Instead the process stays on the run queue. This saves on the overhead involved in moving processes onto and off of the run queue.

Interrupt handlers can't wait on the run queue, but they are able to call `SemLockCritical`. If the semaphore is clear, it is locked, and status code 0 ("ercOK") is returned. If, however, the semaphore is already set, a nonzero status code is returned, and the interrupt handler can take an alternate action.

The following are other significant ways that critical section operations differ from the noncritical ones:

- A *timeout* parameter is not provided.
- There is no use count. Processes cannot lock critical section semaphores repeatedly in nested procedures.
- They are handled differently at termination and can affect the way a process is suspended. (See "Terminating and Suspending Processes.")

`SemClearCritical` is paired with `SemLockCritical` when a process is ready to release the critical section semaphore lock. This operation may cause a process switch if a process waiting for the semaphore has a higher priority than the process calling `SemClearCritical`.

Terminating and Suspending Processes

Critical section semaphores are handled specially when a process is terminated. If a process owns any *nonterminatable* critical section semaphores, the process is not terminated until it clears the semaphores. This is an option that can be specified in the call to SemOpen. (See “Opening a System Semaphore.”)

Critical section semaphores also affect the way a process is suspended. If a process has a critical section semaphore locked, it cannot be suspended. The SuspendProcess and SuspendUser operations increment the suspend count for the process, but the process continues to run until it clears all its critical section semaphores. (For details on SuspendProcess and SuspendUser, see “Process Management Operations” in the section entitled “Process Management.”)

Mutual Exclusion on CTOS

On CTOS, mutual exclusion is intrinsic. It is simply a built-in feature of interprocess communication (IPC) and message passing. (For details on IPC, see the section entitled “Interprocess Communication.”) If your applications make requests of system services, chances are you aren’t aware of (and don’t really need to know) how IPC serializes access to shared resources.

As a brief review, let’s look at the memory manager service. When an application needs to allocate memory, it makes a call such as AllocMemorySL. The request interface builds a request block and sends it to the memory manager exchange. Upon receipt of a message, the memory manager is scheduled to run. When it is the highest priority process in the run queue, it begins processing the first request queued at its exchange.

The memory manager can be interrupted at any time by a process with a higher priority. If a higher priority process calls AllocMemorySL, a request block is again queued at the memory manager exchange.

Regardless of the number of messages queued, the memory manager processes one request at a time until it is done and it responds to that client. Only then does it start processing the next request queued. With no other process accessing memory management data except the memory manager itself, mutual exclusion is assured.

Behind-the-Scenes Use of a Semaphore

It may be interesting to note that, while IPC is enforcing exclusive access to the memory resource by the memory manager's clients, the memory manager itself is using a semaphore to gain access to the very resource its clients want. Let's examine this situation more closely.

CTOS provides a set of memory map operations for accessing and updating the memory map. The memory map must be shared by several CTOS processes, including the memory manager and the paging service. Formerly (on multipartition and variable partition operating systems), the memory map operations were requests. On demand-paged, virtual memory operating systems, however, request-based operation would result in a deadlock in which both the paging service and the memory manager wait at their response exchanges for responses from each other.

Ironically, a semaphore breaks the deadlock. The memory management operations are reimplemented on virtual memory systems as a system-common service. As you recall, system-common services require a semaphore to manage resources.

Implementing a Simple Mutual Exclusion Semaphore

On CTOS, you can implement a simple mutual exclusion semaphore using the Send and Wait primitives. The semaphore can easily be modeled as the exchange, as outlined below:

1. To create the semaphore, allocate an exchange and send a message (any message) to it, for example

```
AllocExch(pSemExchRet)
```

```
Send(semExch, pMsg)
```

2. To acquire the semaphore for mutual exclusion, wait at the exchange, for example

```
Wait(semExch, ppMsgRet)
```

3. Once the semaphore is obtained, perform the required operation on the shared resource (such as updating a queue).

4. To clear the semaphore so the next process can acquire it, send a message to the exchange, for example

```
Send(semExch, pMsg)
```


If a process is waiting for the semaphore, the first process waiting at the exchange runs. If more than one process is waiting, the rest continue to wait until the first process clears the semaphore by calling `Send`. If no process is waiting, the message is queued at the exchange until another process waits at the exchange.

Signaling and Synchronizing Processes

A *signaling semaphore* is used to announce the completion of a single event to one or more waiting processes. Processes don't own signaling semaphores: they just set and clear them. If a process wants to be informed of an event, it sets the semaphore with one operation and waits for the event to take place with another.

Signaling is an entirely different function than mutual exclusion. As such, it has its own set of operations, namely

- `SemClear`
- `SemMuxWait`
- `SemNotify`
- `SemSet`
- `SemWait`

These operations must not be used interchangeably with the mutual exclusion operations. `SemClear` is the exception. It functions in either of two ways: to clear a signaling semaphore or to unlock a mutual exclusion semaphore.

Using the Signaling Semaphore Operations

To set the semaphore a process calls the `SemSet` operation. `SemSet` doesn't check the semaphore lock bit, which may already have been set by another process. It simply sets the semaphore regardless of its present state.

If the process wants to wait for the event, it calls `SemWait`. `SemWait` causes the caller to wait for the semaphore specified by the semaphore handle. If the semaphore is already clear, no waiting is necessary. Otherwise the caller waits at its default response exchange until either the semaphore is cleared with the `SemClear` operation by the process responsible for setting it or a timeout occurs. When the semaphore is clear, `SemWait` allows the caller to continue executing.

If the caller wants to wait for several semaphores to clear and isn't concerned about which clears first, it can call the `SemMuxWait` operation. If none of the semaphores are clear when `SemMuxWait` is called, the caller waits at its default response exchange until either one of the semaphores clears or a timeout occurs. `SemMuxWait` returns the first semaphore that clears.

The `SemNotify` operation can be used to notify the caller that the specified semaphore is clear. It sends a message to this effect to the exchange the caller provides.

***Note:** `SemNotify` can only be used with system semaphores.*

Both of the signaling semaphore waiting operations and `SemNotify` have a *timeout* parameter that allows the caller to control the semaphore waiting semantics.

When the `SemClear` is paired with `SemSet`, it clears the specified signaling semaphore, regardless of its state. Once a semaphore is clear, `SemClear` notifies all processes waiting for the semaphore.

The Producer/Consumer Model

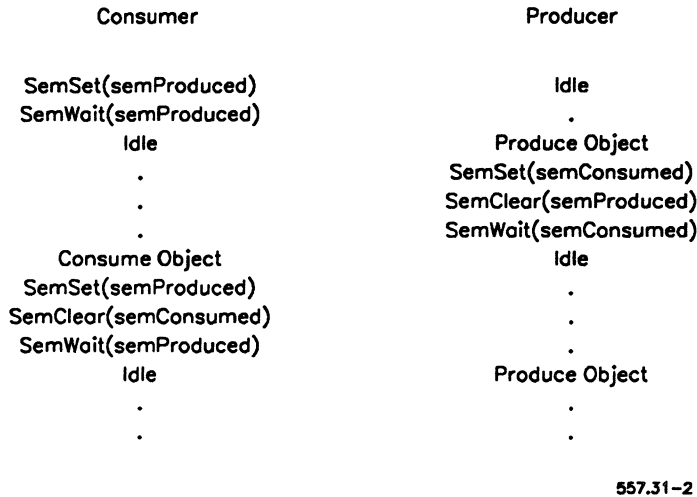
Signaling semaphores are used to notify processes of an event. The event can be thought of as an object produced. The processes waiting for this event are the object consumers.

Consider a typical producer/consumer model. The consumer process waits for an object to be produced. When an object is available, it consumes the object and signals the producer process when it is done. Then it waits for the next object to consume.

The producer process produces an object and signals the consumer process when it is done. The producer waits for the object to be consumed before producing another.

This producer/consumer model can be demonstrated using the signaling semaphore operations, as shown in Figure 31-2.

Figure 31-2. Producer/Consumer Model Using Semaphores



In the figure, the consumer and producer activities are shown in parallel. Note how the semaphore operations work to synchronize execution of the producer and consumer processes. While one process is executing, the other is idle (waits).

Two semaphores are at work: one to produce the object, and the other, to consume it. In the figure, these are named *semProduced* and *semConsumed*, respectively. The actual notification that an object is produced or consumed occurs when SemClear is called with the appropriate semaphore handle.

For example, when the producer calls SemClear with the handle of the *semProduced* semaphore, SemClear notifies the consumer that an object is produced. Several consumer processes might be waiting on the *semProduced* semaphore to consume the object. Likewise several producers might be waiting for the *semConsumed* semaphore to produce another object. In such cases, process priority determines which process gets to run and, therefore, obtain the semaphore.

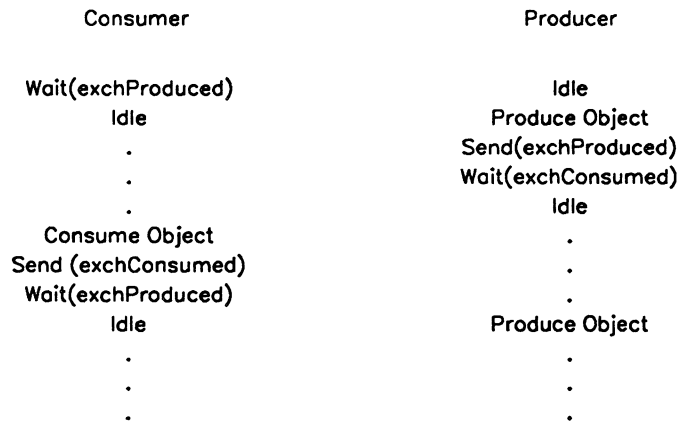
The sequence of producing and consuming operations shown in the figure can take place just once, or it can occur repeatedly, depending on the application design and requirements.

CTOS Primitive Solutions to the Producer/Consumer Model

Although the signaling semaphores may be used to signal and synchronize processes, there are CTOS IPC kernel primitives that accomplish this same function. Figure 31-3 and Figure 31-4 demonstrate how to use IPC primitives to create the producer/consumer model. (For details on IPC, see “Interprocess Communication.”)

CTOS typically accomplishes signaling and process synchronization with the Send and Wait primitives. Using these primitives, the producer/consumer model can be set up as shown in Figure 31-3.

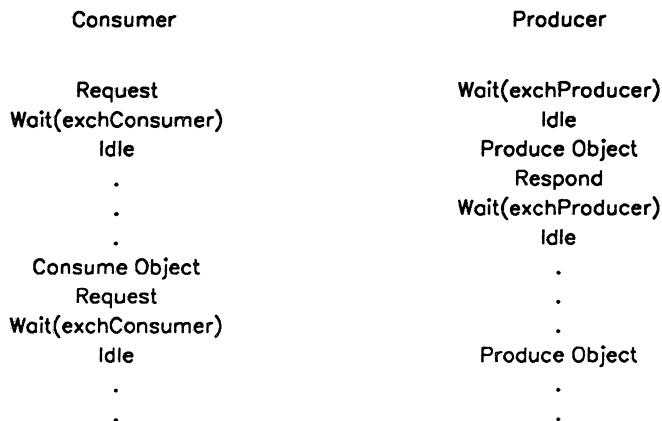
Figure 31-3. Producer/Consumer Model Using Send/Wait



557.31-3

Perhaps the most straightforward producer/consumer model is demonstrated by the CTOS Request and Respond primitives. This is shown in Figure 31-4.

Figure 31-4. Producer/Consumer Model Using Request/Respond



557.31-4

In conclusion, there are CTOS alternatives to the signaling operations. If you use the IPC kernel primitives, you basically let the operating system manage and control process synchronization for your application.

Using RAM Semaphores

Caution

Although CTOS supports RAM semaphores, they should only be used by applications ported to CTOS that *already* use them. New applications needing semaphores should use the operations for creating system semaphores instead.

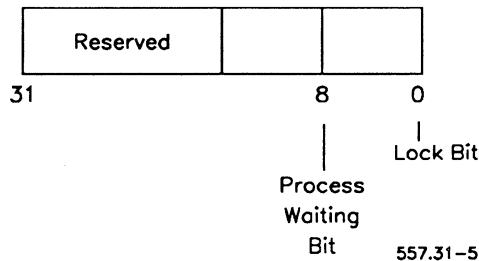
RAM semaphores are used for mutual exclusion and for signaling. The system supports an unlimited number of RAM semaphores.

When an application wants to use a RAM semaphore it simply allocates memory for the semaphore variable (double word) in local memory. The memory address of this variable becomes the semaphore handle, which is provided in calls to the semaphore operations.

RAM Semaphore Variable

A RAM *semaphore variable* consists of a double word in the caller's data space. See Figure 31-5.

Figure 31-5. RAM Semaphore Variable Structure



Initially, the semaphore variable is zeros to indicate that it is unowned. When the caller locks a mutual exclusion semaphore or sets a signaling semaphore, the lock bit is set to 1. If any other processes are waiting for the semaphore, the operating system sets the process waiting bit.

Every time a process waits for a semaphore, this information must be maintained in an internal system record. For system semaphores, the record is allocated only once when the semaphore is opened. For RAM semaphores, however, the record must be allocated each time a process waits for the semaphore. This extra overhead in managing waiting processes results in RAM semaphores being slower than system semaphores. This is just one disadvantage. See “Guidelines to Using RAM Semaphores” for other, more serious caveats to using RAM semaphores.

Guidelines to Using RAM Semaphores

If you must use RAM semaphores, you should be aware of the following caveats:

1. The RAM mutual exclusion semaphore can't be locked recursively in a nested procedure.
2. If a RAM semaphore is being shared by different user numbers, be aware that there is no cleanup of processes owning semaphores at termination.
3. Priority scheduling of processes can be subverted, resulting in uncontrollable busy loops.
4. You can't use certain semaphore operations, namely SemOpen, SemClose, SemLockCritical, SemClearCritical, and SemNotify.

If you are using the RAM semaphore for mutual exclusion, the semaphore can't be locked recursively in a nested procedure. RAM semaphores have no use count to control recursive locking by the same process. If a process tries to lock a semaphore it already owns, this action results in a deadlock. The process is trapped forever waiting for the semaphore to clear.

You can avoid locking a semaphore more than once to prevent the above situation. In some cases, however, you may not have control over a deadlock. Say, for example, a RAM semaphore is shared between different user numbers. If the user currently owning the semaphore is terminated while another is waiting to obtain the semaphore, the waiting user is deadlocked because it is not notified of the semaphore owner's termination.

In the same situation, system semaphores send a status code to the waiting user to inform that user of the death of the semaphore owner. At best, one can avoid sharing RAM semaphores among different user numbers.

System semaphores also have internal structures to monitor and adjust process priorities. RAM semaphores do not. As a consequence, priority scheduling of processes can be subverted with RAM semaphores.

Suppose, for example, three processes (P1, P2, and P3) want to use a semaphore S. Each has a different priority with P1, the highest priority, and P3, the lowest. Circumstances are such that the following sequence of events occurs:

1. P3 locks S.
2. P3 is preempted by P2.
3. P2 is preempted by P1.
4. P1 tries to lock S but must wait for the lock.
5. P2 runs again.

Once P2 starts executing, it can effectively prevent P1 from running indefinitely. This situation can be prevented if, temporarily, the priority of P3 is raised to be the same as that of P1. This allows P3 to finish using S, making S available for P1 to use.

System semaphores adjust process priorities in this way. The priority of the process owning a lock is always adjusted to equal that of a higher priority process waiting for the lock, thereby preventing processes from spinning in busy loops.

Finally, you can only use a subset of the semaphore operations. For example, you don't use the SemOpen and SemClose operations for RAM semaphores. These operations are only used to open system semaphores. Opening system semaphores sets up the appropriate structures to manage them. (This avoids most of the problems just mentioned that occur with using RAM semaphores.)

Neither can you use the critical section operations SemLockCritical and SemClearCritical. SemNotify doesn't work with RAM semaphores either.

Semaphore Operations

The semaphore operations are described below are categorized by use. Operations are arranged alphabetically in each group. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

System Semaphores

These operations are used to open and close system semaphores. System semaphores can be used for either mutual exclusion or for signaling processes.

SemClose

Closes a system semaphore that was opened by a previous call to SemOpen.

SemOpen

Opens the system semaphore with the specified options and returns a semaphore handle. If the semaphore is already open, SemOpen returns the handle of the semaphore originally assigned.

Mutual Exclusion

These mutual exclusion operations are used for noncritical semaphores, where use of processor time is not critical.

SemClear

Clears the specified mutual exclusion semaphore locked by SemLock when the use count is 0. SemClear decrements the use count each time it is called by the same process to lock the semaphore. It may cause a process waiting for the semaphore to be put on the run queue. It may also cause a task switch.

SemLock

Causes the caller to wait until the specified semaphore clears (if necessary); then it locks the semaphore. With each call to lock the semaphore by the same process, SemLock increments the use count.

Critical Section

These mutual exclusion operations are used for critical sections to economize on the use of processor time.

SemClearCritical

Clears the specified critical section semaphore set by SemLockCritical.

SemLockCritical

Causes the caller to wait (if necessary) in the run queue until the specified critical section semaphore clears; then it locks the semaphore.

Signaling and Synchronization

These operations are used for signaling and synchronizing process execution.

SemClear

Clears the specified signaling semaphore set by SemSet. This operation notifies all processes waiting for the semaphore with the SemMuxWait, SemNotify, or SemWait operation. It may cause a process waiting for the semaphore to be put on the run queue. It may also cause a task switch.

SemMuxWait

Causes the caller to wait (if necessary) until one of several semaphores set by SemSet clears or a timeout occurs. If more than one semaphore clears while the caller is waiting, the operation returns the index of the semaphore that clears first.

SemSet

Sets the signaling semaphore specified by the semaphore handle.

SemWait

Causes the caller to wait (if necessary) until the specified signaling semaphore set by SemSet clears or a timeout occurs.

Section 32

Inter-CPU Communication

What is Inter-CPU Communication?

The *inter-CPU communication* (ICC) facility provides communication between CPUs among the different processor boards on a shared resource processor (SRP). ICC is an extension of interprocess communication (IPC). (For details on the types of SRP processor boards and board naming, see “Processor Boards SRP” in the section entitled “Understanding Hardware,” in the *CTOS System Administration Guide*.)

A shared resource processor is compatible with the workstations at the request level. Messages passed between a client and a system service on the same processor board use IPC. The kernel routes the request to the system service exchange; the system service performs its function and responds to the client’s exchange, acknowledging service completion. See “IPC Summary” in the section entitled “Interprocess Communication.” The section describes and illustrates the request/response model on a workstation. (This same model is used for requests routed locally on a single SRP processor board.)

When a client requests a system service, the kernel examines its request routing table to determine, for example,

- If the request block is correctly formed
- To which system service the request is to be sent

These actions are taken in the case of ICC or IPC. However, the destination to which the request is sent determines if the request is handled as a normal IPC message or if it is to be routed by means of ICC.

ICC Terminology

ICC involves *interboard routing* or the passing of the request and the response message between processor boards. To accomplish this, ICC uses

- Processor boards identified by *slot numbers*
- Routing information contained in the operating system *request routing table*
- Communication between processors over a high-speed *bus*
- *W-block*, *Y-block*, and *Z-block buffers* for storing copies of request blocks
- *Bus addresses* of data buffers
- A CPU description table (*CDT*) on each processor board containing the address of that board's ICC segment
- A request and response *ring queue* in the ICC segment
- Free *W-*, *Y-*, and *Z-block ring queues* in the ICC segment
- A *doorbell interrupt*

Slot Numbers

At the hardware level, each processor in a system is identified for ICC communications by a unique 8 bit *slot number*. Slot numbers range from a high of 77h to a low of 20h.

The slots in the base enclosure are numbered 70h to 77h. As viewed from the back of the enclosure, 70h is the leftmost slot, slot 77h the rightmost. The enclosure closest to the base enclosure has slots 60h through 67h, the next enclosure in the line has slots 50h through 57h, and so on. (For details on slot numbering conventions, see "Diagnosing Problems" in the section entitled "Troubleshooting," in the *CTOS System Administration Guide*.)

The slot number is used by certain operating system operations to identify a particular processor and by the hardware to accomplish interboard addressing.

You can use the `GetProcInfo` and the `GetSlotInfo` operations to retrieve such hardware information and, thereby, explicitly control ICC routing. You would use these operations if using one of the shared resource processor routing types (described below) is not sufficient.

Shared Resource Processor Routing Types

Table 32-1 describes each of the shared resource processor routing types used to route requests between SRP processor boards.

The shared resource processor routing type routes a request to the target processor board. As Table 32-1 shows, network routing (that is, routing involving a handle or specification), combined with the shared resource processor routing type, determines the final destination of requests routed by the routing types *rLocal* and *rDevice*. (For details on network routing, see “Routing Requests” in the section entitled “Interprocess Communication.”)

The way you define the routing types when writing a system service to be run on a shared resource processor is described in the section entitled “System Services Management.”

Table 32-1. Shared Resource Processor Routing Types

Field	Description
rLocal*	<p>If the request (block) does not contain a network routing specification for a remote board, the request is served locally. The service exchange is looked up in the service exchange field of the operating system request routing table.</p> <p>If the request contains a specification for a remote board, the shared resource processor filter process calls RequestRemote to route the request to the board specified in the name table in the master processor (disk controlling processor, which can be a GP+SI, FP, or DP). The filter uses the specification name (such as Win1 or FP00) in square brackets as the key to locate the corresponding board slot number in the table. The specification must contain a name in square brackets or a status (error) code is returned, terminating routing.</p>
rRemote*	<p>The request is sent by exchange to the local or the remote board.</p>
rDevice*	<p>If the request contains neither a handle nor specification (no network routing), the request is routed to all boards previously accessed by this user number. If the request contains a handle, it is routed by the handle to the appropriate board.</p> <p>If the request contains a specification for a remote board, the shared resource processor filter process calls RequestRemote to route the request to the board specified in the master processor name table. The filter uses the specification name (such as Win1 or FP00) in square brackets as the key to locate the corresponding board slot number in the table. If the specification does not contain a name in square brackets, the kernel on this board attempts to send the request to a local system service.</p>
rBroadcast	<p>The request is routed to every processor board booted on the shared resource processor.</p>

continued

*This type is frequently used.

Table 32-1. Shared Resource Processor Routing Types (cont.)

Field	Description
rMasterFp	The request is routed to the master processor.
rFileId	The request is routed to the appropriate board by the slot number contained in the first byte of the request block control information.
rLineNum	The request is routed to the Cluster Processor (CP) that handles this line. This routing type is used by the operation MegaframeDisableCluster. Each CP has two lines. For example, CP00 has lines 1 and 2; CP01 has lines 3 and 4; and so on.
rHandle	The request is routed to the target file processor (GP+SI, FP, or DP) using an indexed field in the handle.
rMasterCp	(Unused)

Blocks

Blocks are areas of memory allocated for cluster or ICC communications. X-blocks are used to transfer information over cluster lines.

W-blocks, Y-blocks, and Z-blocks are used to transfer ICC messages. To support the programs that you will run, you can configure your operating system for the size and number of blocks you need. A *Z-block* is used if the message can fit into a small number of bytes; otherwise, a *Y-block* is typically used. *W-blocks* are used in special cases for the transfer of unusually large amounts of data, such as database information, when configuring Y-blocks to be of this size would be constraining on your system.

The *CTOS System Administration Guide* provides detailed information on configuring W-, X-, Y-, and Z-blocks for maximum system performance.

CPU Description Table

Each processor in a shared resource processor contains a CPU description table (CDT). The CDT describes the processor to other processors and contains some routing information. For the purposes of ICC, the most significant field in the CDT is the address of the ICC segment for that processor.

ICC Segment

There is an ICC segment that controls ICC communications for each processor. The ICC segment contains

- A ring queue of free W-block buffers
- A ring queue of free Y-block buffers
- A ring queue of free Z-block buffers
- The bus addresses of the buffers
- A ring queue of inbound requests in process
- A ring queue of inbound responses in process
- A buffer ownership table identifying the processor boards currently using buffers
- Tables to mediate buffer wait conditions

Ring Queues of Block Type Buffers

Each ring queue of free W-, Y-, and Z-blocks contains only the addresses of the buffers. The actual buffers are located elsewhere in system memory (possibly outside the ICC segment).

Bus Addresses

Bus addresses are used to send data to another processor board. The contents of a request block, for example, are copied to the service board using a bus address. The bus address and physical address can be different values on processors with hardware that maps bus addresses, for example the 386 processor on a GP SRP board. (For details on bus addresses, see the section entitled “Bus Address Management.”)

Ring Queues of Requests and Responses

The inbound request and response ring queues contain information to help the ICC kernels locate requests and responses.

The *request ring queue* contains buffer numbers. A *buffer number* is used to locate the W-, Y-, or Z-block containing a client's request in the service board's memory.

The *response ring queue* contains response numbers. A *response number* identifies the system service's slot number, the address of the W-, Y-, or Z-block (containing a copy of the request block) in the service board's memory, and the address of the client's request block.

Buffer Ownership Table

The *buffer ownership table* identifies processors currently using buffers on this board. If a processor board crashes, this table is consulted to free any outstanding buffers associated with the crashed board.

Buffer Wait Tables

Sometimes a client must wait for an available W-, Y-, or Z-block buffer on a service board. To mediate buffer wait conditions, the kernel consults the three types of tables.

Each processor board contains one wait exchange table, one wait-satisfied flags table, and three wait flags tables, one for each type of data block (W, Y, and Z). Each of these tables contains 36 entries, one for each possible processor board on a shared resource processor. For details on how these tables are used, see "Mediating Buffer Wait Conditions."

Doorbell Interrupt

Each processor in a shared resource processor can send an interrupt, called a *doorbell interrupt*, to any other processor board in the system.

For example, during inter-CPU communication, the kernel on a processor board sends a doorbell interrupt to alert the kernel on the target processor board that a request or response has been registered in a ring queue and, thus, needs processing.

Interboard Routing

Each processor board can send and receive requests and responses. In the description of interboard routing that follows, each of the following actions is described separately:

- Sending a request
- Receiving a request
- Sending a response
- Receiving a response

Sending a Request

When a client calls Request, the kernel uses the request code in the request block as an index into the operating system routing table. The routing table determines the destination of the service exchange. (For details, see “Routing Requests” in the section entitled “Interprocess Communication.”)

Local Routing?

If routing information indicates that the request is to be served locally and a local server exists, ICC is not used. Instead, the request is routed using the normal procedures of IPC.

Note that the memory address of a request served locally is a logical memory address.

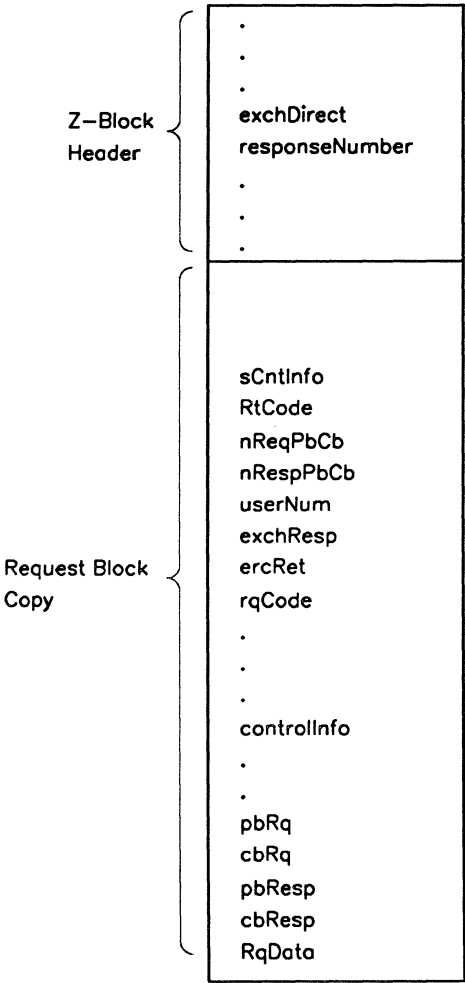
Off-board Routing?

If routing indicates that the request is to be served off board, ICC is used to send the request to the specified service board.

To send the request to the service board, the kernel on the client board does the following:

1. Calculates the size of the request by examining the size of the client's request block memory. Calculation is based on the byte counts of the request block header, the control information, and the request and response data buffers. The kernel uses the size to determine which kind of block-type buffer (W-, Y- or Z-block) to reserve in the service board's memory. (This description of interboard routing shows the Z-block as an example, although any of the ICC block-type buffers could have been used.) Figure 32-1 shows the format of the Z-block.
2. Copies the client's response number into the Z-block header. When the response comes back to the client board, the client board kernel can use this number to obtain the service board slot number and the addresses of the Z-block and the client's request block. (See "Receiving a Response.")
3. Copies the request block contents into the Z-block following the header.
4. Enters the Z-block buffer number into the service board request ring queue. From the buffer number, the Z-block can be located in the service board's memory. (See "Receiving a Request.")
5. Sends a doorbell interrupt to the kernel on the service board.

Figure 32-1. Z-Block



557.32-1

Receiving a Request

The doorbell interrupt from the sending board alerts the kernel on the service board that it has received an off-board request in its request ring buffer. To receive an off-board request, the kernel does the following:

1. Inspects its request ring queue for entries.
2. Obtains the Z-block buffer number from the request ring queue. The kernel uses the number to locate the Z-block buffer.
3. Constructs new pointers to the request and response data in the Z-block. Pointer calculation is based on the count of bytes in the request block header, the control information, and the request and response data. (New pointers must be created because the ones from the client board are always invalid on the service board.)
4. Calls either the Request or the RequestDirect kernel primitive. If a value is specified for the field *exchDirect* in the Z-block header (see Figure 32-1 in “Sending a Request”), RequestDirect can be called with this value (the service exchange) and the memory address of the Z-block as parameters. Calling RequestDirect routes the request directly to the service exchange. Otherwise, Request is called with the Z-block address and the procedure described in “Sending a Request” is repeated.

Sending a Response

A response to a request originated off-board must be sent back to the client board.

The kernel on the service board recognizes a response to be routed off-board by the request block response exchange number. To send a response back to the client, the kernel does the following:

1. Obtains the client's response number from the Z-block header and enters this number into the client's response ring queue.
2. Sends a doorbell interrupt to the kernel on the client board.

Receiving a Response

The doorbell interrupt from the service board alerts the kernel on the client board that it has received an off-board response in its response ring buffer. To receive the off-board response, the kernel does the following:

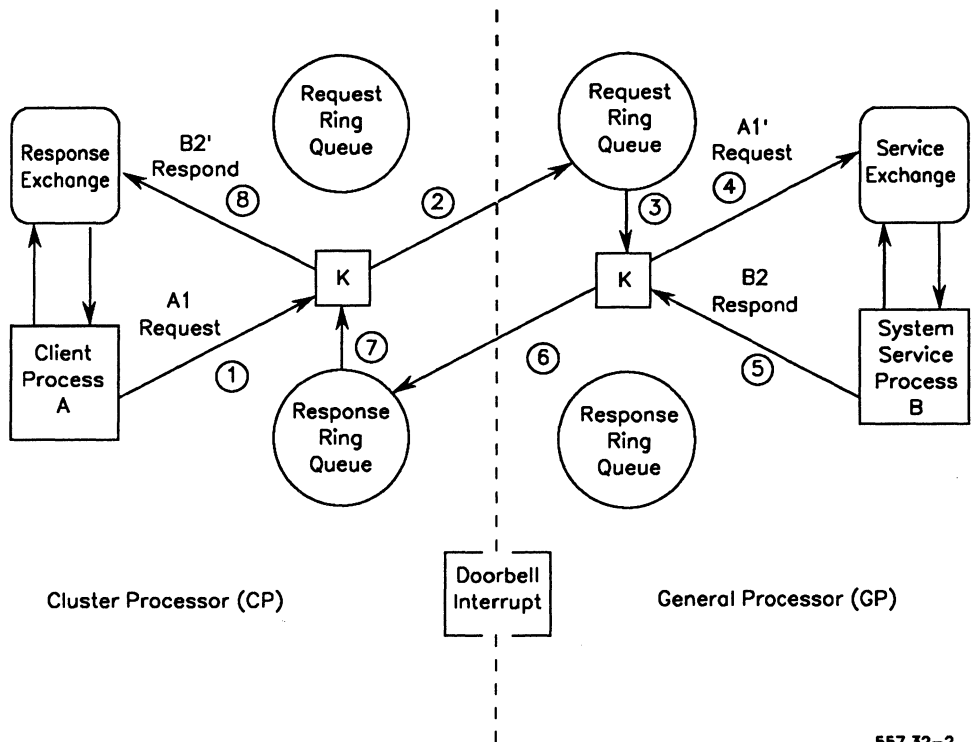
1. Inspects its response ring queue for entries.
2. Obtains the client's response number from the response ring queue and uses this number to find the system service's slot number, the address of the Z-block in the service board's memory, and the address of the request block in the client's memory.
3. Copies the response buffer and the status code from the Z-block to the client's memory.
4. Frees the Z-block holding the copy of the client's request block in the service board memory.
5. Calls Respond, providing the memory address of the request block.

Sending and Receiving Messages

Figure 32-2 shows the interaction of client A on a Cluster Processor (CP) board and system service B on a General Processor (GP) board. Circled numbers 1 through 4 in the figure generally show the path over which the request travels from client A to system service B. Numbers 5 through 8 show the response path back to client A.

In the figure, client A on the CP board requests (A1) a service provided by system service B on the GP board. The kernel on the CP board copies the request block contents to a Z-block in the GP processor, enters the Z-block buffer number into the GP board request ring queue, and rings the GP's doorbell.

Figure 32-2. ICC Interaction



557.32-2

The kernel on the GP board locates the Z-block using the Z-block buffer number, calls Request (A1'), and sends the request to system service B's service exchange. System service B processes the request and responds (B2).

The kernel on the GP board acts on the Respond (B2) by entering the response number into the client board response ring queue and rings the CP's doorbell.

The kernel on the CP board copies the response back to client A's request block, frees the Z-block buffer on the GP board, and calls Respond (B2').

Of the two kernels involved in ICC, the client kernel takes on the more active role in managing ICC data structures. It is the client kernel's responsibility to calculate and reserve an appropriate sized data buffer in the service board's memory, copy the request block to this memory, and free the buffer when it is no longer needed. The service board kernel merely constructs local pointers to the data in the buffer.

In the figure, note that Request and Respond function in two ways. One invocation of Request and Respond send information to another board; another invocation of Request and Respond wait at an exchange.

Mediating Buffer Wait Conditions

Sometimes it is necessary for a client to wait for a buffer on the service board when none is available in the ring queues of free buffers. To mediate buffer wait conditions, the kernel consults the following types of tables on each processor board:

- One wait exchange table
- Three wait flags tables (one for each type of data block)
- One wait-satisfied flags table

Each table contains 36 entries, one for each possible processor board on a shared resource processor.

In the wait exchange table, each entry is an exchange corresponding to a processor board. The memory address of a request block is queued at the appropriate exchange when a data block is not available for the client. For example, when a client on board A must wait for a buffer on service board B, the kernel on board A queues the memory address of the client's request block (pRq) at the exchange for board B in its own wait exchange table. (See Figure 32-3.) In the figure, if more than one client were waiting for a block on the same board, each request block would be linked to the previous one already queued at the exchange.

Figure 32-3. Board A's Wait Exchange Table

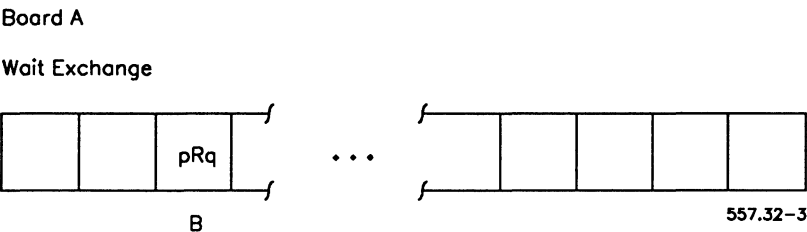
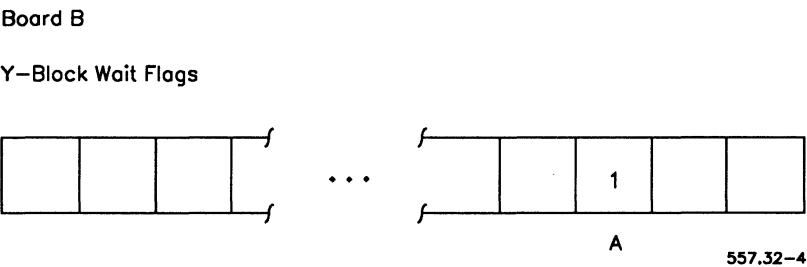


Figure 32-4. Board B's Y-Block Wait Flags Table

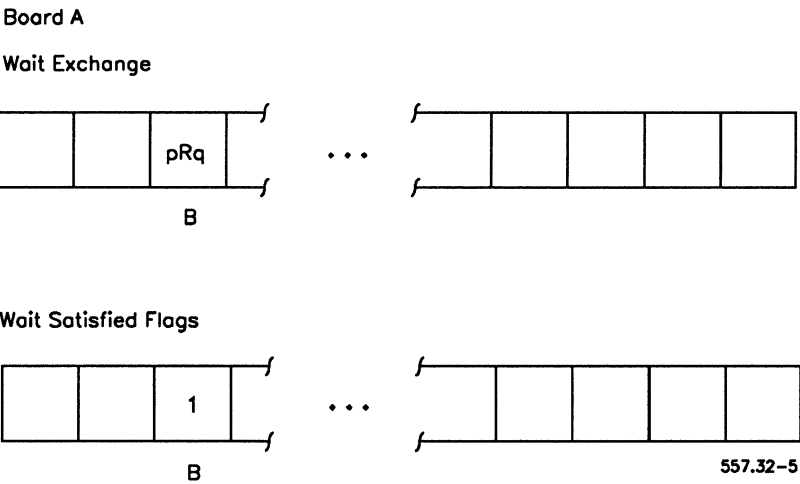


For each type of data block, there is a 36 bit wait flags table. Each bit in this table is a flag corresponding to a processor board. The kernel on the client board sets these bits when buffers are not available. For example, if the kernel on client board A determines that the client needs a Y-block on board B and none is available in board B's ring queue of free Y-blocks, the kernel on board A sets the flag corresponding to board A in board B's Y-block wait flags table. (See Figure 32-4.)

On each board there is, in addition, a 36 bit wait-satisfied flags table. Each bit in this table is also a flag corresponding to a processor board. The table is used to indicate blocks that have been freed. For example, if a kernel on board C frees a Y-block on board B, the board C kernel searches board B's Y-block wait flags table in round robin fashion to find the next board (if any) waiting to use the block. If the kernel finds board A waiting (as shown in Figure 32-4), the kernel sets the corresponding bit for board B in board A's wait-satisfied flags table and rings board A's doorbell.

Figure 32-5 shows the table entries on board A.

Figure 32-5. Wait Exchange and Wait-Satisfied Flags tables



The flags in the wait-satisfied flags table are merely indicators that there are blocks available for reuse. Board A's doorbell interrupt routine actually checks all flag entries in the wait-satisfied flags table. (Figure 32-5 is a simple case showing just the flag set for board B.) For each board with a Wait Satisfied flag set, the routine clears the corresponding flag in the appropriate wait flags table and retries the first request queued in the wait exchange table. The routine repeats this procedure for all requests queued at the exchange until the exchange is empty or there are no more free blocks.

The kernel uses the following guidelines to mediate buffer waits:

- The kernel always selects a data block that best fits the request. A Y-block would not be selected for a request that can fit in a much smaller Z-block, for example.
- The kernel handles requests in the order that they arrive for processing.
- If a request is waiting for a buffer on a specified board, all subsequent requests for that same board must wait for the first request to be serviced.

As these guidelines point out, heavy use of one type of data block can severely impair system performance if an insufficient number of blocks of that type are configured.

ICC Operations

The ICC operations are described below. Operations are arranged in alphabetical order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

GetBoardInfo

Accepts the slot number of an SRP processor board and a case value. Based on the case value specified, information is returned about the specified processor in the memory area provided.

GetProcInfo

Returns the name of the processor on which the caller is running.

GetSlotFromName

Accepts the name of an SRP processor board and returns the corresponding slot number.

GetSlotInfo

Determines the slot numbers of other processors in a shared resource processor system.

QueryBoardInfo

Returns information about the specified processor in the memory area provided based on the specified case value. For backwards compatibility, the *GetBoardInfo* library procedure can be used instead of this request to obtain the same information.

ReadKeySwitch

Returns a word value that indicates the keyswitch position in which a shared resource processor was booted.

RemoteBoot

Causes another dormant processor to be bootstrapped with a specified System Image.

RequestRemote

Requests a system service from a remote board on a shared resource processor by sending the request through ICC to the remote board.

Section 33

System Services Management

What is System Services Management?

This section describes a type of program that serves requests for system resources. The section begins with an overview of how a service program operates. In essence, this is a review of interprocess communication (IPC). The section then unveils how such service programs, upon dynamic installation, become indistinguishable from the rest of the operating system. In the course of this discussion, you are provided the basic guidelines on how to write the components of a system service program as well as how to convert an existing service such that multiple instances of it can be run on a shared resource processors (SRP). In addition, the section describes, through example, the consequences of writing a system service incorrectly.

For examples of how to write system services, see “Writing System Services for the XE-530” and “Asynchronous System Service Model” in the *CTOS Programming Guide*. This section provides the necessary background for reading those sections. Although the asynchronous model described in the *CTOS Programming Guide* is a variant of the type of system service described in this section, it shows how the guidelines for writing a system service can be implemented in code.

System Service Terminology

A *system service* is a program that provides services to other programs. Examples of the types of service provided include opening and closing disk files, sending output to a printing device, or accepting input from the keyboard. A service can manage access to a resource, such as a file or a printer. The program requesting the service is called a *client*. Any program, including another system service, can be a client.

On shared resource processors, more than one instance of a service may be installed on different SRP boards. In this case, every instance of the service can be accessed by a client anywhere in the system.

Multi-instance services are particularly useful for communications in which one instance of the service provides too few lines, or one SRP board provides too little processor power to run many lines at a high speed.

Note: *On protected mode operating systems, system services must execute in protected mode.*

Overview of Operation

A system service does not communicate with a client directly. Instead, correspondence is by means of *interprocess communication* (IPC). In the following description of system service operation, some of the IPC concepts (described in detail in the section entitled “Interprocess Communication”) are summarized.

Note: *Although this section is dedicated to the more familiar type of system service that is request-based (uses IPC), a second type of service exists that is not. In the section entitled “System-Common Services Management,” the two types of service are compared.*

A system service receives IPC messages from clients. The message is a special IPC message called a request block.

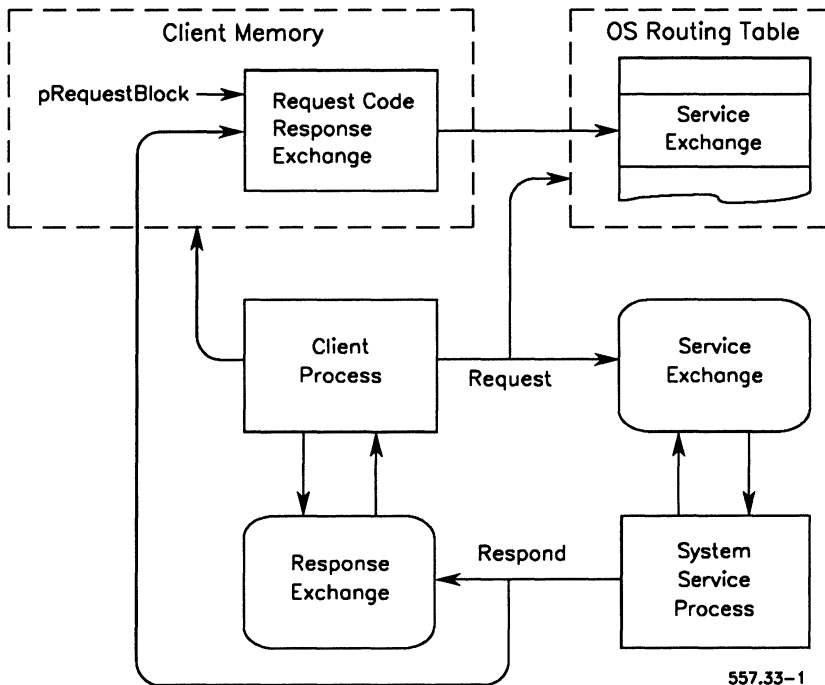
A *request block* is a data structure containing the specification and the parameters of the chosen system service. The request block includes fields for the request code and the client’s response exchange in addition to other fields that describe the request. (For details, see “Request Block Format” in the section entitled “Interprocess Communication.”)

The *request code* is a 16 bit value that uniquely identifies the desired system service. For example, the request code for the OpenFile operation is 4.

A request code is used both to route the request to the exchange of the appropriate system service and to specify which of its several functions the request is for.

The system service waits at its service exchange until it receives a request block from a client. (See Figure 33-1.)

Figure 33-1. Client and System Service



557.33-1

The client uses either of two methods to send a request block to the system service's exchange. The client can

- Use the request procedural interface, which builds the request block and calls `Request`
- Call `Request` directly, in which case the client builds its own request block

`Request` signals the kernel to examine its request routing table. The kernel uses the request code as an index into the table to locate the system service's exchange.

Upon receipt of a request block, the system service verifies the information it contains.

If the information is valid, the system service performs its service and answers the client's request by filling in the request block with its response and a status code value of 0 (meaning no error occurred or "ercOK"). If the request is invalid, however, it places a status code indicating the error in the request block.

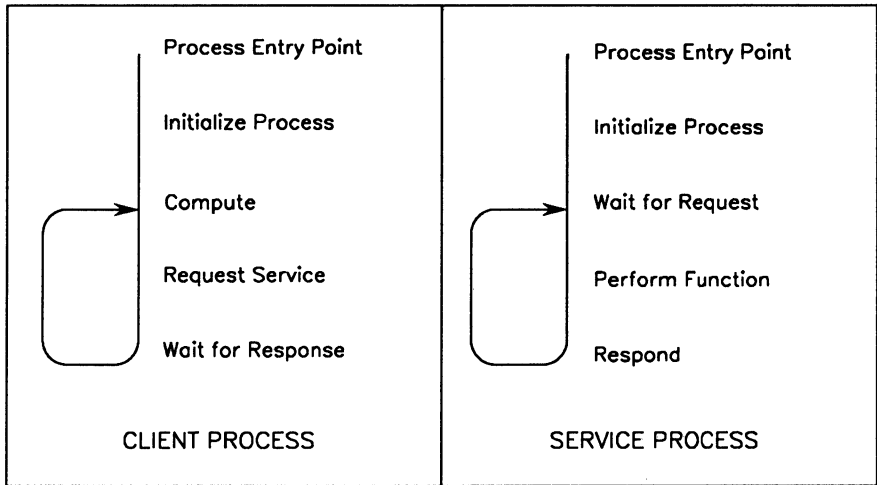
Upon completion of these functions, the system service calls `Respond`. `Respond` routes the request block back to the client's exchange as specified in the request block.

Figure 33-2 compares the program model of a system service to that of a client.

In the figure, the system service initializes. Then, it spends its time waiting. Upon receipt of a request block from a client, the system service processes the message and then loops back to its wait.

This is a different model than that of a normal application program. An application spends its time computing, waiting only as required for a service to be performed so it can continue computing.

Figure 33-2. Processing Flow



557.33-2

Built-In System Services

A built-in system service is one that is linked into the system image so that it is present when the operating system is bootstrapped. Examples include the file system and the keyboard service.

The differences between the various types of operating systems are a function of the built-in services each has to offer. A cluster workstation operating system, for example, includes a workstation agent. A cluster workstation with a local file system includes a file service as well. (See “Workstation Operating Systems” in the section entitled “Overview of Operating System Concepts.”)

Dynamically Installable System Services

A dynamically installable system service is one that can be added to the system image without regenerating the operating system. This type of system service is created as an application program. It becomes part of the operating system during its initialization.

The CD-ROM System Service and Mouse Services are examples of installable system services. You also can write your own installable services. (See “Guidelines for Writing a System Service.”)

Dynamically installable services extend operating system functionality. You can install and deinstall them at any time without altering the system in any way. While installed, they function in the same way as built-in system services.

Request Routing Table

The operating system contains a request routing table for its built-in system services. Request routing tables are used by the kernel to determine where to send request blocks.

An entry in the request routing table typically includes the following information about a request:

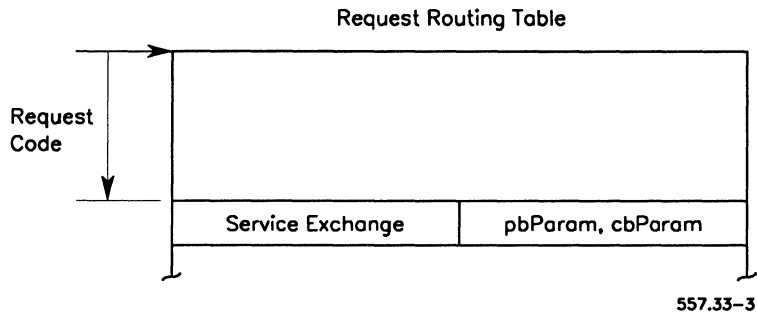
- The request parameters
- The system service exchange of the requested system service

The kernel uses the request code as an index into the table to locate the system service exchange. (See Figure 33-3.)

When a system service is dynamically installed, the request routing table is extended.

You may decide, for example, to install an electronic mail service at your cluster workstation. The electronic mail package updates the request routing table to reflect its service exchanges.

Figure 33-3. Request Routing Table Fields



System Service Package

In its simplest form, a dynamically installable system service package consists of two software components: the request definitions for the system service and the system service itself.

To allow updating of the request routing table, each of these components is designed in a special way.

Requests

The *request definition* includes the request code, the request parameters, the system service exchange, and various other fields. (For details, see “Guidelines for Defining Requests.”)

The requests served are defined in a *loadable request file*. The contents of this loadable request file are merged with the contents of the system request file, *Request.sys*. The merge occurs during installation of the system service onto the system disk.

When bootstrapped, the operating system reads *Request.sys*, loads it into memory, and adds the new requests to the request routing table.

By reading *Request.sys*, the operating system thus receives acknowledgment that the new requests exist. The operating system sets the service exchange field for each new request according to the request file.

The System Service

After the operating system is bootstrapped, the system service also is loaded into memory. This is usually done by an entry in the *SysInit.jcl* file. (For details, see the section entitled “Installing System Services” in the *CTOS System Administration Guide*.)

As part of initialization, the system service calls `ServeRq` for each request it will serve. `ServeRq` updates the service exchange field (in the request routing table) for each request code to reflect the system service exchange.

If the system service is to be able to deinstall itself later or if it is a filter, it must call `QueryRequestInfo`, which determines the exchanges to be served, before calling `ServeRq`.

A *filter* substitutes its exchange for that of another system service. (See “Guidelines for Writing a System Service” and “Filters.”)

Guidelines for Writing a System Service

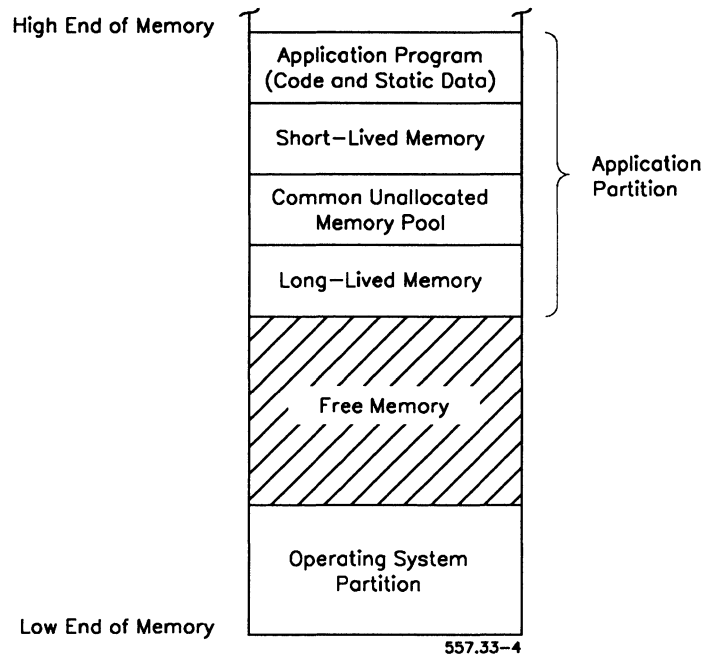
The guidelines for writing a system service (below) outline the order in which certain system operations should be called by a program converting to a system service. What happens to memory before and after the conversion is also illustrated.

Note: *All system services must be installed before a context managing program can be loaded into the application partition (primary partition) in memory. (For details, see the section entitled “Partitions and Partition Management.”)*

Initialization and Conversion to a System Service

A system service begins as an application program when it is first loaded into memory. Its user number is associated with all the partition components shown in Figure 33-4.

Figure 33-4. Before Conversion to a System Service



The typical operating sequence of a system service initializing itself and converting to a system service is described as follows:

1. Use `ChangePriority` if desired. A system service priority normally should be in the range of 10 to 64.
2. Use all required initialization operations, such as `AllocExch`, `AllocMemorySL`, and `CreateProcess`, to get required resources before converting to a system service. On virtual memory operating systems, however, this is not a requirement. The service can allocate or deallocate global linear address space at any time. (For details, see “Using Global Linear Address Space,” in the section entitled “Memory Management.”)

3. Use the `QueryRequestInfo` operation to find out the current exchanges for all of the requests to be served. This is required if the system service is to be able to deinstall itself later or if the system service is going to filter messages destined to other system services. (For details, see “Deinstallation of a System Service” and “Filters.”)
4. Optionally use the `SetMsgRet` operation to provide the exit run file with an informative message indicating success or failure of the installation.
5. Use the `ConvertToSys` operation to become part of the operating system.

Caution

Before calling `ServeRq` to serve a request, a filter must call `QueryRequestInfo` to see if the request is already being served by another system service. Otherwise, the exchange of the original system service will be overwritten.

6. Use the `ServeRq` operation for each request code to be served. In addition, the `ServeRq` operation must be used for each system request to be filtered. (See “System Requests.”) Note that it is best not to use the default response exchange or else the server will be unable to use the request procedural interface.

If more than one instance of the system service is to be installed on a shared resource processor, use the `UpdateFpMountTable` operation to update the master processor name table with the request routing name. The name (portion of the routing specification in square brackets) controls the target board where the request will be served. (Additional guidelines for multi-instance system services are summarized in “Converting to a Multi-Instance Service.”)

7. Use the `Exit`, `ErrorExit`, or `Chain` operation to reload the exit run file into memory. Note that since the program is now a part of the operating system, these calls will return to the new system service (for normal application programs, these calls never return).
8. Use the `SetPartitionName` operation to set an identifiable (up to 12 character) name for the system partition. `SysServiceXX` is the default name, where `XX` is the user number.

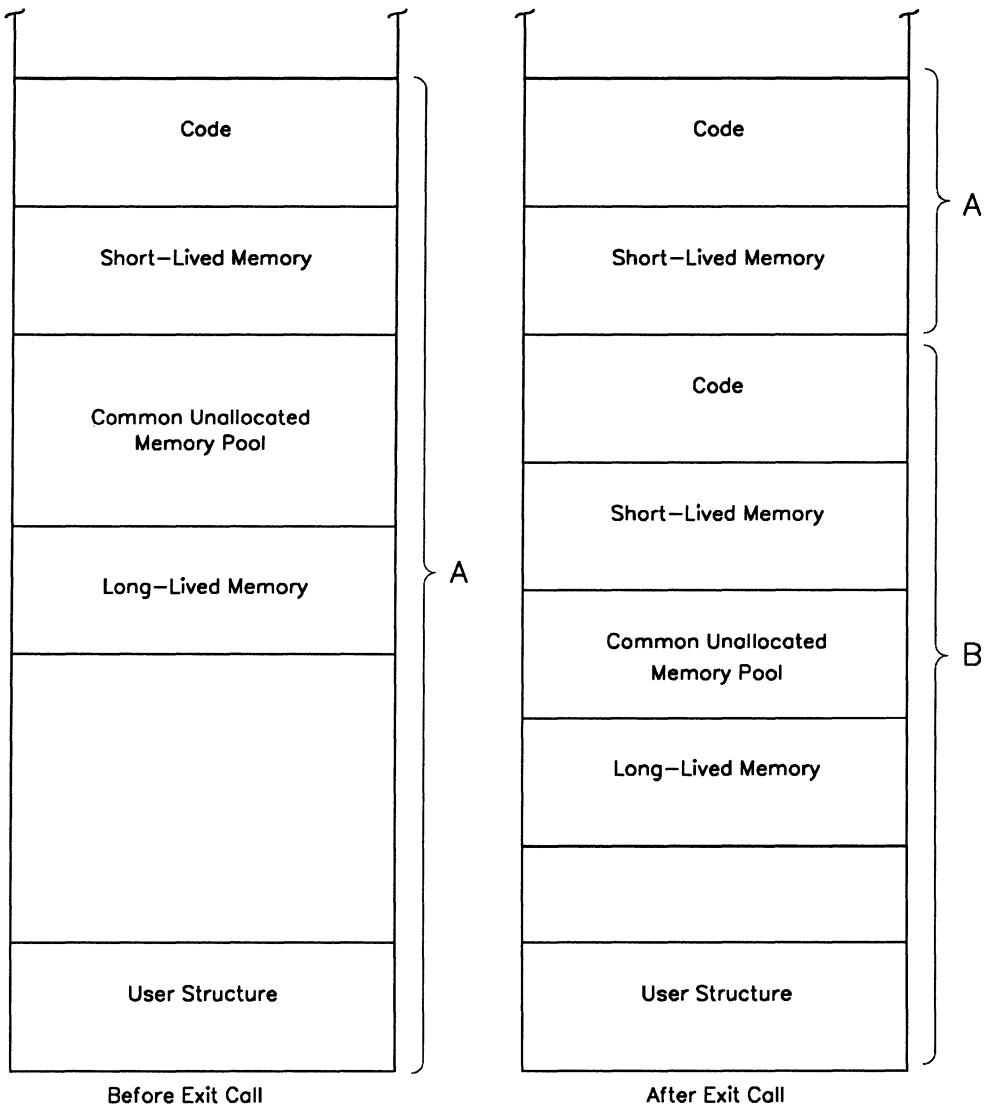
Figure 33-5 compares memory before and after the exit operation is called, as described in guideline 7. In the figure, **A** represents the partition components (short-lived memory, long-lived memory, and so forth) associated with the program converting to a system service. **B** represents the partition components associated with the exit run file in the new primary (application) partition created.

If, for some reason, the system service attempts to call an `Exit` operation a second time after calling `ConvertToSys`, the operating system records this information in the system Log File.

Note: *A program can call `ConvertToSys` as long as memory consists of a single primary partition as shown in B in the figure; otherwise, status code 810 (“Invalid request”) or status code 206 (“Invalid user number”) is returned.*

Partition components are described in detail in the section entitled “Partitions and Partition Management.” For now, note that the components of the system service partition are only a subset of those from the original partition (shown on the lefthand side of the figure before the call to `Exit`). Furthermore, on real mode operating systems, the system partition is assigned a new user number. (For details, see “Change User Number Requests.”)

Figure 33-5. Conversion to a System Service



557.33-5

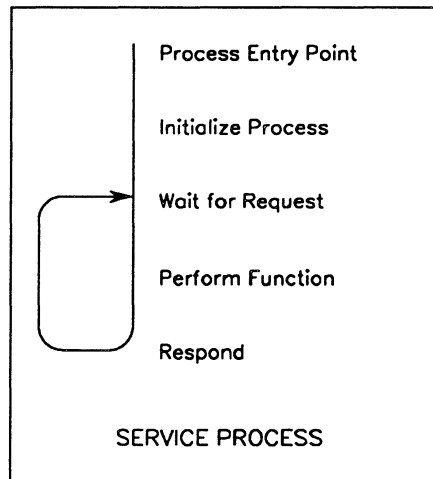
System Service Main Program

The program model of a system service is shown in Figure 33-6.

After initialization and conversion to a system program, the system service enters its main program. In the main program, it calls Wait and waits at its exchange. This gets the system service into its normal state: waiting to do work.

The loop in Figure 33-6 signifies the program instructions the system service executes when it performs a service for a client. After executing these instructions, the system service calls Respond and loops back to its waiting state.

Figure 33-6. System Service Program Model



557.33-6

Restrictions and Requirements of Operation

As part of the operating system, a system service is a special type of program. After a successful call to `ConvertToSys`, it must adhere to the following specific rules to function correctly:

- On multipartition and variable partition operating systems, it must not allocate or deallocate memory. (On virtual memory systems, however, the system service can allocate dynamic memory out of the global linear address space.)
- It cannot write to the video, read from the keyboard, or call `ErrorExitString`.
- If it is controlling DMA on virtual memory systems with paging enabled, it must use the operations for DMA mapping to obtain buffers that will be locked into contiguous physical memory. (For details, see “Bus Address Management.”)

All system services must perform some common functions in the system by serving termination, abort, and swapping requests. (For details, see “System Requests.”)

Guidelines for Defining Requests

The information needed for defining a request is contained in the special Standard Software text file, *RequestTemplate.txt*. Table 33-1 describes the fields in this file. You can also use a different file called *Request.0.asm*, which is supplied with earlier versions of Standard Software.

Either file works. *RequestTemplate.txt*, however, is friendlier:

- It is a text file you can edit.
- You use the Make Request Set utility program, which is much faster than the assembler (used with *Request.0.asm*) and provides more comprehensive error checking. (For details on Make Request Set9, see the *CTOS Executive Reference Manual*.)

System services for a shared resource processor must serve either local or global requests. *Local requests* are served on the same processor board as the system service. *Global requests* are served on any SRP processor board. Shared resource processor system services must serve requests of the same routing type. (See “Shared Resource Processor Routing Types” in the section entitled “Inter-CPU Communication.”)

Table 33-1. RequestTemplate.txt Fields

Field	Description
:RequestCode:	Uniquely identifies the request. (For details, see "Request Codes" in the section entitled "Interprocess Communication.")
:RequestName:	Identifies the request to a user. This entry is optional but strongly recommended.
:Version:	Indicates whether a request has been updated (default = 0). Requests are generally not updated.
:LclSvcCode:	Is used by the operating system for a special case but is not generally used in writing system services (default = 0).
:ServiceExch:	Indicates the exchange to which the request is routed. This field is changed when a system service calls the ServeRq operation, which provides the exchange to which the request is routed.
:sCntlInfo:	Indicates the number of bytes of control information (default = 6).
:nReqPbCb:	Indicates the number of request pb/cb pairs.
:nRespPbCb:	Indicates the number of response pb/cb pairs.
:Params:	Defines the request procedural interface. This field is used by the operating system for validation of request blocks.
:NetRouting:	Describes routing by resource handle and by specification.
:SrpRouting:	Describes how requests are routed among boards on a shared resource processor. (For details on the values of this field, see "Shared Resource Processor Routing Types" in the section entitled "Inter-CPU Communication.")
:WsAbortRq:	Is the request code for the abort system request.*
:TerminationRq:	Is the request code for the termination system request.*
:SwappingRq:	Is the request code for the swapping system request.*
:ChgUserNumRq:	Is the request code for the change user number system request (multipartition operating systems only).*

*For details, see "System Requests."

Creating Loadable Request Files

To create a loadable request file, use either the *RequestTemplate.txt* or the *Request.0.asm* template file. Using these templates to create a loadable request file is described below:

1. Copy the template to a file identifying the system service.
 - If you use *RequestTemplate.txt*, copy the template to a file, such as *RequestServer.txt*.
 - If you use *Request.0.asm*, copy the template to a file, such as *Request.X.asm*, where *X* is an alphanumeric character in the set (0..9, A..Z) that identifies a group of requests for the system service.
2. Use a text editor to edit your file according to the instructions provided.
3. To build the request file,
 - If you used *RequestServer.txt*, run your text file through the Make Request Set utility. Make Request Set reads your text file, checks for errors, and creates a binary file, *RequestServer.bin*.
 - If you used *Request.0.asm*, assemble and link your file to create the binary file, *Request.X.sys*.
4. Use the Install New Request utility to merge your request(s) with the system file, *Request.sys*. (For details on Make Request Set9 and Install New Requests9, see the *CTOS Executive Reference Manual*.)
5. Bootstrap the operating system. Bootstrapping results in the operating system reading the single system request file, *Request.sys*, and adding the loadable requests to the request routing table.

Table 33-2 compares and summarizes the templates.

Table 33-2. Creating a Loadable Request File

RequestTemplate.txt	Request.0.asm
Copy template to <i>RequestServer.txt</i>	Copy template to <i>Request.X.asm</i>
Edit the text file	Edit the macros
Use Make Request Set	Assemble and link to build <i>Request.X.sys</i>
Use Install New Request to merge your request(s) into the single system file, <i>Request.sys</i>	Use Install New Request to merge your request(s) into the single system file, <i>Request.sys</i>
Bootstrap system	Bootstrap system

System Requests

System requests are issued by the operating system to system services. These requests notify system services of clients that are terminating or being swapped to a disk file.

The system requests are of the following types:

- Termination
- Abort
- Swapping
- Change user number

Each of these request types is described in the following paragraphs. (Examples of how to define these requests are contained in the file *RequestTemplate.txt*.)

Termination and Abort Requests

Termination and *abort* requests function similarly in that they notify system services that clients have terminated. Upon notification, system services can release resources, such as open files and locked Indexed Sequential Access Method (ISAM) records, allocated to the terminating clients.

The operating system issues termination requests whenever a client terminates for the following reasons:

- The client called Chain, Exit, or ErrorExit.
- A user pressed ACTION+FINISH.
- A partition managing program called the TerminatePartitionTasks operation to terminate the client.

In addition to termination requests, the operating system issues abort requests at a server under the following circumstances:

- When the server detects that it cannot communicate with a cluster workstation
- When a partition is vacated with the VacatePartition operation or through lack of an exit run file

These requests are issued for the following reasons:

- To ensure that no requests will be returned to the program after it has been terminated and replaced in memory by another program
- To inform servers that resources allocated to the program should be freed

System services must respond to outstanding requests before responding to termination or abort requests. Although a terminating client does not need the response, certain operating system structures the client was using, such as Z-blocks for interboard routing on a shared resource processor, must be made unavailable for future use.

Termination Request to the File System

The following is an example of how the file system service uses the termination request. The example also indicates the consequences of a file system not calling ServeRq to serve a termination request.

When a user initiates the **Copy** command in the Executive, the Executive makes requests to the file system to read and write files to disk.

During execution of these requests, the user presses the key combination, **ACTION+FINISH**. This terminates the Copy program and results in the operating system issuing a termination request to the file system process.

In response to the termination request, the file system process terminates any outstanding read or write requests initiated by the Copy program.

If the file system did not serve the termination request, the Copy program's exit run file, the Executive, would be reloaded into memory. An outstanding Write request responded to by the file system process at this time would result in the response data being written to the Executive's memory rather than to the Copy program's memory.

Swapping Requests

Swapping requests are issued to system services whenever the operating system is going to suspend a program and swap it to disk. Swapping requests ensure that no responses are made to clients in a program that is not resident in memory.

Note: *Swapping requests are obsolete on demand paged, virtual memory operating systems.*

When a system service receives a swapping request, it is required to respond to all outstanding requests with the same client user number and then to respond to the swapping request.

The system service must respond immediately using one of the following strategies:

- It can hold the swapping request until all outstanding requests for the client are completed. If the service has completed its outstanding requests, it should respond immediately to these requests and then respond normally to the swapping request.
- It can respond to all outstanding requests for the client with status code 37 (“Service not completed”). The operating system intercepts this special response status code and the program is swapped to disk. Later, when the program is swapped back into memory, the operating system reissues the original outstanding requests to the system service.

It is transparent to the program that it is being swapped out of memory or that any of its requests are being handled other than in the usual manner.

Change User Number Requests

Change user number requests are required only by multipartition (real mode) operating systems. On these systems, the primary partition is always user number 1. When the new system partition is created at ConvertToSys (as described and illustrated in “Initialization and Conversion to a System Service”), a new user number is assigned to the system partition to maintain user number 1 as the primary partition user number. As a result, change user number requests are issued to all existing system services to notify them of the new number.

Converting to a Multi-Instance Service

On a shared resource processor, instances of a system service can run on multiple boards simultaneously. To convert a system service to a multi-instance system service, make the following changes to it:

1. Change the routing type for the requests to be served to *rDevice*. (See “Shared Resource Processor Routing Types” in the section entitled “Inter-CPU Communication.”)
2. All requests to be served (except system requests) must be either routed by handle or routed by specification. (For details on these routing styles, see “Routing by Handle” and “Routing by Specification” in the section entitled “Interprocess Communication.”)
3. The shared resource processor filter process termination request (request code 12383) must be added to the loadable request files of all workstations that access the system service as well as to the shared resource processor system file, *Request.sys*.
4. Establish the routing name for the system service using the *UpdateFpMountTable* operation.

Because resource handles are shared by all boards of a shared resource processor, a global handle table maps a unique handle to each user number in the system. System services can use the bits in the handle for any purpose. Applications, however, must not depend on the particular numeric values of handles returned except for use in routing requests to the system service.

Filters

A *filter process* is a system service that is interposed between a client and a system service process that operate as though they were communicating directly with each other. The filter does this by substituting its exchange for that of the original system service in the operating system request routing table.

If your system service acts as a filter, it can intercept requests intended for another system service, and either service them itself, or reissue them to the original service after performing filtering or some other function.

Types of Filters

Filters are of three types: replacement, one-way pass-through, and two-way pass-through. Each type is described in the paragraphs that follow.

Replacement

A *replacement* filter intercepts requests (using the Wait or Check kernel primitive), performs a service based on the intercepted request, and then responds to the request. In this case the filter replaces the original system service.

One-Way Pass Through

A *one-way pass-through* filter intercepts requests and then sends the request on to the system service exchange. It uses the ForwardRequest kernel primitive to forward a request block to a system service for further processing.

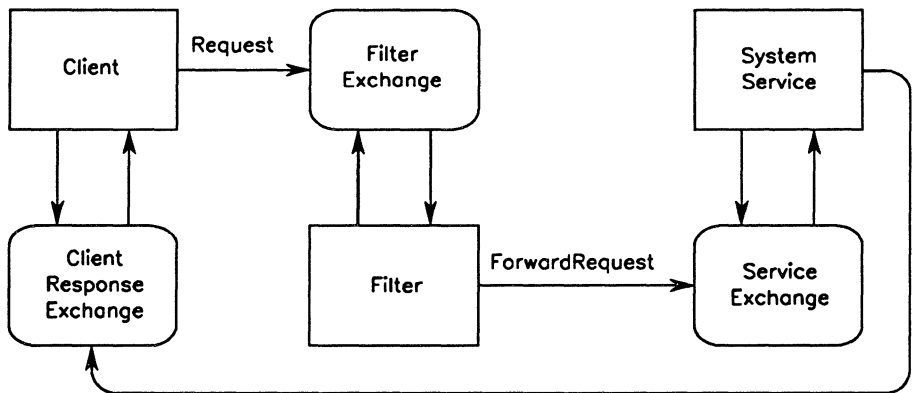
Figure 33-7 shows how this type of filter is used.

The following sequence of events is shown in the figure:

1. The client issues a Request.
2. The filter proceeds from its Wait.
3. The filter issues ForwardRequest (or Send) to the original system service's exchange.
4. The system service proceeds from its Wait.
5. The system service calls Respond.
6. The client proceeds from its Wait.

To be compatible in protected mode, one-way pass-through filters must use ForwardRequest, instead of Send.

Figure 33-7. One-Way Pass-Through Filter



557.33-7

Two-Way Pass Through

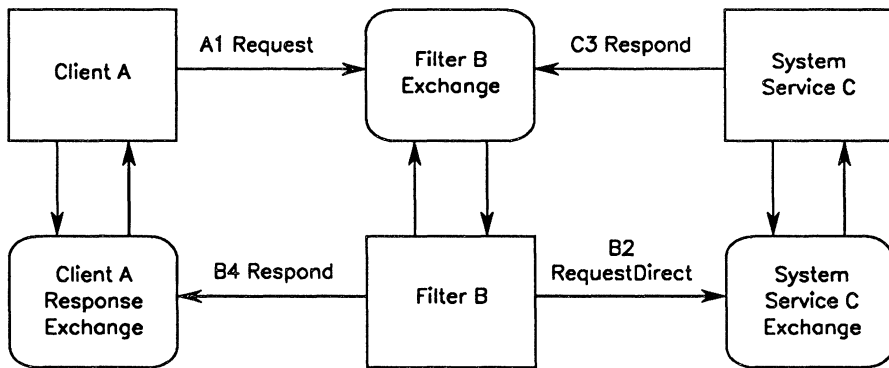
A *two-way pass-through* filter intercepts requests and reissues them to the original system service exchange using the RequestDirect kernel primitive. It also intercepts the Respond and responds back to the client.

Figure 33-8 shows how this type of filter is used.

The following sequence of events is shown in the figure:

1. The client issues a Request.
2. The filter proceeds from its Wait.
3. The filter changes the `exchResp` field to its own exchange and then issues RequestDirect to the original system service's exchange, then Wait.
4. The system service proceeds from its Wait.
5. The system service issues a Respond.
6. The filter proceeds from its Wait.
7. The filter changes the `exchResp` field back to the client's exchange and issues a Respond.
8. The client proceeds from its Wait.

Figure 33-8. Two-Way Pass-Through Filter



557.33-8

System Requests for Filters

A filter that uses only the replacement method should have its own system requests for termination, abort, and swapping. (For details, see “System Requests.”) In this case the filter is the same as a normal system service.

A filter process that uses one of the pass-through methods of filtering must filter the system requests of the original system service(s). If the filter uses the two-way pass-through method for any requests, it also must use that method for the system requests.

Use of Filters

Filters can be used in many ways. A filter, for example, might be used between the file management system and its client process to perform special password validation on all or some requests. Filters are commonly used by the keyboard service to filter keystrokes for various accounting purposes.

Workstation agents and Net agents act as filters in directing IPC messages to other destinations for further IPC processing.

Results of Not Serving Swapping Requests

The following example describes the consequences of a keyboard filter not performing a `ServeRq` on keyboard swapping requests.

Context Manager maintains an outstanding `ReadActionKbd` request to the keyboard manager to receive `ACTION` key combinations. The key combination, `ACTION+NEXT`, for example, alerts Context Manager to switch to a different context (user number).

The two-way pass-through filter has been installed to intercept the `ReadKbd` requests.

Under Context Manager, a user is running an Executive program as the current context. The Executive is issuing a series of ReadKbd requests while the user is typing characters onto the command line. The user types the characters C, O, and P, followed by the key combination, ACTION+NEXT.

Context Manager, whose priority is higher than the Executive, receives the ACTION+NEXT key combination before the filter receives the P. In response, Context Manager initiates a swap to bring in the chosen context.

A swapping request is issued by the operating system. The request bypasses the filter and goes directly to the keyboard process, which responds.

The filter, which was not notified of the context switch, holds onto the ReadKbd request. As a result, the swap file fails with status code 813 ("Cannot swap out this partition").

Deinstallation of a System Service

A system service may deinstall itself. To do this, you must write a utility program that runs at the same workstation as the system service and that issues a deinstallation request to the system service.

The deinstallation request should have the user number of the system service as one of the response parameters. Deinstallation should follow these steps:

1. The utility program issues a deinstallation request to the system service.
2. The system service performs a ServeRq on all of its requests to restore them to their original values.
3. The system service checks all of its exchanges and internal queues and responds to all requests it may still have, except the deinstallation request.
4. The system service calls SetPartitionLock(0) to unlock its partition.
5. The system service calls GetUserNumber to find its user number.

6. The system service copies its user number to the memory address of the deinstallation request response field and then responds to the request with status code 0 (“ercOK”) in the ercRet field.
7. The system service calls Wait and waits for the removal of the partition at one of its exchanges.
8. The utility program receives the response to its request. If the ercRet field is 0 (“ercOK”), it calls VacatePartition followed by RemovePartition, using the user number returned by the system service.

System Service Operations

The system services management operations described below are categorized as basic or system requests. Operations are arranged alphabetically in each group. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Basic Requests Used by all System Services

ConvertToSys

Converts all processes, short-lived memory, and exchanges in an application partition to system service processes, system memory, and system exchanges, respectively, in a system partition.

QueryNodeName

Obtains the node name of the local node where this request is issued.

QueryRequestInfo

Determines the exchange to which a request and its local service code are routed.

ServeRq

Is used by a dynamically installed system service process to declare its readiness to serve the specified request code.

SetPartitionName

Changes the name of the caller's partition.

System Requests

System requests include termination, abort, swapping, and change user number requests.

Section 34

System-Common Services Management

What is a System-Common Service?

System-common procedures are procedures within the operating system that are available for use by programs. The Video Access Method (VAM) is an example of a collection of such procedures for programming to the video device. System-common procedures have been available in all operating system versions. These existing procedures are described in the *CTOS Procedural Interface Reference Manual*, along with the standard operating system library procedures and kernel primitives. With the introduction of protected mode operating systems, however, system-common procedures can be written by the programmer and installed dynamically at any time. (See Appendix A, “Operating System Features,” to determine which systems support this feature.)

To the programmer, dynamically installable system-common procedures are very similar in function to the system services described in “System Services Management.” By convention thus far in this manual, *system service* has meant a system program that uses request-based interprocess communication (IPC) to provide services to client programs. This section introduces a broader meaning to the term system service in which dynamically installable system-common procedures are a second system service option. Hereafter, to distinguish between the two types of system service, system services that use IPC are called *request-based system services*. Services that provide system-common procedures, on the other hand, are called *system-common services*.

System-Common Service Model Overview

The system-common service model is different from the request-based model in one major way.

In the request-based model, the client sends a request to the service. The service then takes over control of the processor and performs its work. While the system service executes, the client process waits. When the service has finished, it sends a response to the client. This allows the client to regain control of the processor and begin executing again.

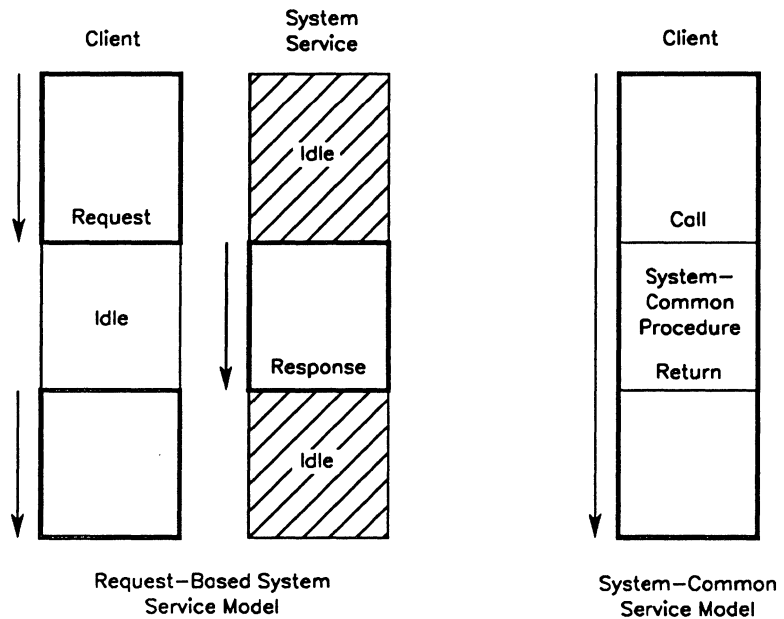
In the system-common model, there are no requests or responses. When the system-common service installs itself, it tells the operating system the entry point and parameters of each of its system-common procedures. After it has defined its system-common procedures to the operating system, the service simply waits to deinstall. It performs no more work itself.

To use a system-common procedure, clients simply call the procedure as if it were part of the client program. The operating system detects that the called procedure is actually a system-common procedure, and transfers the client's execution point to the beginning of the system-common procedure.

The client never gives up control of the processor. The system-common procedure executes as part of the client process. Then, when the system-common procedure returns, the operating system resets the client's execution point to the appropriate place in the client program.

Figure 34-1 compares the thread of execution in a request-based system service to that of a system-common service.

Figure 34-1. Request-Based Compared to System-Common



557.34-1

Compared to Request-Based System Services

The following compares system-common to request-based services in some detail and provides you with guidelines for deciding which service is more appropriate for a given task.

Similarities

System-common services are similar to request-based system services in the following ways:

- Both types of system service have two complementary components: client code and system service code. The client needs a service performed. The system service performs that service. The VAM service, for example, consists of a collection of system-common procedures, such as PutFrameChars, QueryFrameChar, and FillFrame, that can be called by a client to carry out certain video services. The file system is a request-based service that provides file services to clients that make requests, such as OpenFile, Read, and Write. In either case, the operating system Kernel acts as the dispatcher in seeing that the client is matched with the appropriate service.
- Clients can call system-common procedures using a procedural interface in the same manner as requests are called using the request procedural interface. (Clients of request-based services can, in addition, call the Request Kernel primitive and build a request block for asynchronous operation. For details, see “Using the Kernel Primitives Directly,” in the section entitled “Interprocess Communication.”)
- Initialization code of the system service informs the operating system of where to route the request, for request-based services, or the location of the system common procedure to be called, for system-common services. Request-based system services call the ServeRq operation for each request to be served; whereas, system-common services call the InstallSystemCommon operation for each system-common procedure they contain.
- On protected mode operating systems, both types of system service execute in protected mode. (System-common procedures, in addition, must be based in the global descriptor table.)

Differences

System-common services differ from request-based system services in the following ways:

- System-common services eliminate the need to define requests in loadable request files. (For details on loadable request files, see “Creating Loadable Request Files” in the section entitled “System Services Management.”)
- System-common services may not be deinstalled. Existing applications may have references to the procedures.
- System-common services do not use the request mechanism for communication between the system service and the client. Instead, a system-common procedure executes within the client process, potentially changing all the client’s registers.

For request-based system services, the operating system performs a context switch from the client process to the system service process. (See the section entitled “Process Management,” for details.) On protected mode systems, this context switch is called a *task switch*. (Task switch is an Intel microprocessor term. It is described in the Intel manuals listed in “Related Documentation,” at the beginning of this manual.) The impact of a context (or task) switch is that the client process state (contents of all the client’s registers) is saved before the system service executes and is restored again when the client executes.

- In protected mode, all request-based system services use aliasing to map addresses between the client and the system service. (For details on aliasing, see the *CTOS Programming Guide*.) Because system-common services execute in the context of the client process, they need to use aliasing only for clients executing in real mode.
- Request-based services can be routed within a cluster or over the network and can use inter-CPU communication (ICC) for the interboard routing on a shared resource processor. System-common services, on the other hand, can be used only at the local workstation.

- System-common services must be reentrant. Reentrant simply means that, when a system-common procedure is called, the procedure can be interrupted at any point, then called by another client before services are completed for the first client.

Certain precautions must be taken to ensure code is reentrant. Usually this means ensuring that local variables are on the stack. If any global variables are modified, they must be done so atomically. Remember, when you write a system-common procedure, you are using the client's DS and SS. If your system-common service intends to use DS to access system-common global variables, it must explicitly save away the current contents of DS (the client data segment selector) and load DS with its own data segment selector. Then, before the system-common procedure returns, it must restore the client DS.

A problem can arise when a compiler (such as the PL/M compiler), assumes SS and DS are the same for the medium model of computation. When the compiler creates a pointer to a local stack variable, it uses the contents of DS as the selector. Such a pointer is invalid: the offset correctly points to the local stack variable but the selector is that of the system-common procedure's data segment. If the selector is used, it could generate a GP fault or cause the wrong data to be accessed. (You can avoid this problem in PL/M by not using the `at` symbol as an operator to generate a pointer to a local variable.) To create a valid pointer, SS must be used to create the selector.

Request-based system services need not be reentrant because they are never called. Although some request-based services are designed to process multiple requests simultaneously, most request-based services completely finish processing one request before checking to see if a new request has arrived at the service exchange. In either case, a request that arrives at the exchange is processed when the request-based service is ready for it. This control over when to begin processing a new request is a key difference between a request-based and a system-common service.

Choosing a System Service to Write

Each type of system service has its advantages and disadvantages. If the system designer intends to write a program to be used over the network, a request-based service must be written. If, however, performance is an issue, system-common services are the better choice. This is because system-common procedures do not involve the overhead of a task switch or aliasing for each client. There is some aliasing overhead with system-common services, but this occurs only when real mode clients execute on a protected mode operating system. In this situation, the real mode addresses of the client require mapping to the system-common service, which executes in protected mode. Table 34-1 compares the two types of system service.

Table 34-1. System Service Comparison

Criterion	Request-Based System Service	System-Common Service
Type	Context (or Task) switch	*Call Gate
Installation	ServeRq	SystemCommonInstall
Aliasing	Yes	RMOS clients only
Routing	Cluster, network, ICC	Local only
Code	Need not be reentrant	Reentrant

*Call gates are Intel protected mode structures. For details, see the Intel manuals listed in "Related Documentation."

System-Common Service Writing Guidelines

The following are general guidelines for writing system-common services. The system-common service

1. Must execute in protected mode. This is true of all system services that execute on protected mode operating systems.
2. Must be written using reentrant code. (For details, see the discussion of reentrant code in “Differences.”)
3. Must be based in the global descriptor table (GDT).
4. Must include initialization code that calls `SystemCommonInstall` for each system-common procedure it contains. `SystemCommonInstall` informs the operating system of the system-common procedure’s location and its parameters.
5. Usually calls `ConvertToSys` and `Exit` unless designed to execute in the primary partition. Context Manager, for example, does not call `ConvertToSys`. Instead, it remains in the primary partition to manage other partitions. (For details on `ConvertToSys`, see “Initialization and Conversion to a System Service” in the section entitled “System Services Management.”)

Initialization is the only function the system-common service is required to perform. If, during initialization, no requests are to be served, `KillProcess` can be called to free the process control block (PCB) for use by other processes. Although additional functionality is left to the program designer, some suggestions are described next.

Serving Requests

If the system-common service is to serve requests, it must establish itself as a request-based system service. As part of this procedure, the system service must call `ServeRq` and wait on an exchange. (For details on how to write a request-based system service, see “Guidelines for Writing a System Service” in the section entitled “System Services Management.”)

Serving a Deinstallation Request

If, in addition, the system-common service is to be able to be deinstalled, it must have saved the state of the original system-common procedures at the time of initialization. This can be done as follows:

1. When `SystemCommonInstall` is called to install a system-common procedure, the address of and parameters to the previous system-common procedure are returned to the caller in the *InfoRet* structure. (For details, see the description of `SystemCommonInstall` in the *CTOS Procedural Interface Reference Manual*.) This data must be saved so it can be restored at deinstallation.
2. To deinstall each system-common procedure, call `SystemCommonInstall` again. Provide the saved address of the previous system-common procedure and the procedure's parameter definitions as the *pProcedure* and the *pbParamDef/cbParamDef* parameters, respectively, to the call.
3. Respond to the deinstallation request. (For details on deinstallation, see "Deinstallation of a System Service" in the section entitled "System Services Management.")

Do not have the system-common service call `Exit`. Existing applications may depend on references (established at application load) to the system-common procedures. Calling the `Exit` operation removes the procedures from memory.

How to Write a System-Common Service

Details and an example of how to write a system-common service are contained in the following file, which is shipped on the distribution media for Standard Software:

[Sys]<Sys>UserSysCommonLabel.asm

In addition, you must use this file to assign a number to each system-common procedure, as described below.

Following is a summary of steps on how to write your system-common service:

1. Write one (or more) system-common procedures for the system-common service.

2. Assign each procedure a system-common number. The user range of system-common procedure numbers is 24576 through 32767. To obtain a number that will not conflict among users, however, you can reserve a number in the range of 16384 through 24575 by contacting Unisys, Distributed Systems Division, San Jose Product Support.
3. When writing your initialization routines, call the `SystemCommonInstall` operation for each procedure that is part of your service.
4. Compile the system service. Then link it using an appropriate link command. (For details on linking, see the *CTOS Programming Utilities Reference Manual: Building Applications*.) Be sure that you enter the appropriate option for a GDT-based program in the field *Run File type* of the command form.
5. Create a command to load the system service run file into memory, or add the run file name to the *SysInit.jcl* file at the workstation where the system service is to be used.
6. Edit the file *UserSysCommonLabel.asm* by adding a macro call for each system-common procedure contained in the system service. The macro is of the form

`%OsSubLab(ProcNumber, ProcName)`

where

ProcNumber

Is the number you assigned to the procedure.

ProcName

Is the name of the system-common procedure.

This macro creates an absolute PUBLIC symbol for the system-common procedure name. When a client program using the system-common procedure is compiled and linked, the value of this macro is resolved to the correct address of the procedure.

7. Assemble *UserSysCommonLabel.asm* to create the object module *UserSysCommonLabel.obj*.

Calling the System-Common Service

To use a dynamically installed system-common procedure in a client program, perform the following steps:

1. In the client code, declare the procedure name **EXTERNAL**.
2. Link the application with *UserSysCommonLabel.obj*.
3. Client programs using any of the collection of sytem common procedures of the system-common service need only call the intended procedure by its name in a program statement. This is the standard way of calling any system-common procedure whether it already exists in the operating system or if it is dynamically installed.

To Eliminate Linking

The system service writer optionally can use the Librarian to place the *UserSysCommonLabel.asm* file in a standard operating system library. (See the *CTOS Programming Utilities Reference Manual: Building Applications* for details on the Librarian.)

Doing so eliminates the need for the programmer using the system-common procedure(s) to link the module explicitly with the client program.

System-Common Service Operations

The system-common service operations are described below. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

SystemCommonInstall

Informs the operating system of the parameter list and address of the system-common procedure. `SystemCommonInstall` can be called only by GDT-based programs executing in protected mode.

SystemCommonQuery

Returns information about the requested system-common entry.

Section 35

Extended System Services

What are Extended System Services?

The operating system can be extended just by dynamically installing system services. Dynamically installable services are called *extended system services*.

This chapter lists and briefly describes the operations for the following such services:

- Asynchronous System Service Model
- CD-ROM Service
- Command Access Service
- Mouse Services
- Performance Statistics Service
- Queue Manager
- Sequential Access Service
- Spooler
- Voice/Data Services™

For details and examples of how to use these services in your programs, see the *CTOS Programming Guide*. The extended system service operations and system structures are described in detail in the *CTOS Procedural Interface Reference Manual*.

Extended System Services Operations

The extended system service operations are described briefly below. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Asynchronous System Service

AllocMemoryInit

Allocates memory out of the data segment (DS) space below the last code segment.

AsyncRequest

Uses the Kernel primitive Request to send the request block to the operating system for routing to the appropriate service exchange.

AsyncRequestDirect

Uses the Kernel primitive RequestDirect rather than Request to send the request block to a specified exchange.

BuildAsyncRequest

Sends a request to the exchange serving the request.

BuildAsyncRequestDirect

Sends a request to the specified exchange.

CheckContextStack

Validates the context stack.

CreateContext

Allocates stack space for a context from the heap.

HeapAlloc

Allocates memory from the heap.

HeapFree

Returns memory to the heap.

HeapInit

Initializes the heap.

LogMsgIn

Logs the message pointed to by the global variable *pRq*.

LogRespond

Logs the response to the request pointed to by the global variable *pRq*.

LogRequest

Logs the request pointed to by the global variable *pRq*.

ResumeContext

Resumes execution of a context at the point where execution left off the last time the system service made an asynchronous request to an external agent.

SwapContextUser

Handles swapping requests.

TerminateAllOtherContexts

Terminate each active context except for the calling context at system service deinstallation.

TerminateContext

Terminates a context and return its stack space to the heap.

TerminateContextUser

Responds to all client requests for a given user number. Transparent to the caller, it waits for all outstanding asynchronous system service requests to return.

CD-ROM Service

Volume Information

CdDirectoryList

Returns a list of all directory paths for a CD-ROM disc.

CdGetDirEntry

Returns directory record information, in either ISO or High Sierra format, for a specified path.

CdGetVolumeInfo

Returns specific information for a particular volume (the primary volume descriptor, copyright file name, abstract file name, and bibliographic file name).

CdSearchClose

Closes an open CD-ROM search session. The caller provides CdSearchClose with the search handle of the session to be closed.

CdSearchFirst

For a given file specification (possibly with wild cards), returns the directory record and path information for the first matching file on the disc.

CdSearchNext

If wild cards were used in a previous CdSearchFirst operation, CdSearchNext finds additional files that match the wild card specification.

CdVerifyPath

Verifies the existence of the specified path.

Status

CdControl

Controls disc eject and door locking mechanisms. In addition, returns device status, location of head, sector size, and volume size.

Read File

CdClose

Closes the specified open CD-ROM file or device.

CdOpen

Opens an existing CD-ROM file or device and returns a file or device handle.

CdRead

Transfers a specified number of bytes from the CD-ROM device to a caller-specified memory area.

Audio

CdAudioCtl

Controls the audio playback channels in the CD-ROM disc drive, including selection of output channels and volume setting, and control of audio playback, pause, and stop. Also provides information on the disc as a whole, on particular tracks, and on the current Q-channel address.

Other

CdAbsoluteRead

Reads the specified number of sectors from the CD-ROM device, and writes the sector contents to a caller-specified memory area.

CdServiceControl

Deinstalls the CD-ROM Service.

Command Access Service

ObtainAccessInfo

Provides access information about the current user such as whether the user is permitted to use a particular Executive command. The user name and the specified command name must appear in the access file *[/Sys]<Sys>UserCmdConfig.sys*.

ObtainUserAccessInfo

Provides access information about a specified user. This operation differs from *ObtainAccessInfo* in that it allows the caller to obtain information about any user, not just the current user.

Mouse Services

Setup

PdInitialize

Resets the mouse routines.

PdSetCharMapVirtualCoordinates

Defines a virtual coordinate space over the current character map window normalized coordinate space.

PdSetCursorType

Indicates the cursor type (character or graphics).

PdSetMotionRectangle

Defines a motion rectangle in virtual screen coordinates.

PdSetMotionRectangleNSC

Defines a motion rectangle in normalized screen coordinates.

PdSetTracking

Turns cursor tracking on and off.

PdSetVirtualCoordinates

Defines a virtual coordinate space over the normalized coordinate space.

Queries

GetIBusDevInfo

Returns information about the nonkeyboard I-Bus device.

PdGetCursorPos

Returns the current cursor coordinates in virtual screen coordinates.

PdGetCursorPosNSC

Returns the current cursor coordinates in normalized screen coordinates.

PdQueryControls

Allows an application to query various pointing device controls.

PdQuerySystemControls

Allows an application to query various pointing device system-wide controls.

ReadInputEvent

Returns one input event from an input device.

ReadInputEventNSC

Returns one input event from an input device in normalized screen coordinates.

Cursor Shape

PdLoadCursor

Loads the cursor bit map if a graphics cursor has been requested.

PdLoadSystemCursor

Loads the default pointing device cursor bit map.

PdReadCurrentCursor

Copies the current cursor bit map for the application to the specified address.

PdReadIconFile

Reads the graphics cursor icon file.

Cursor Movement

PdSetCursorDisplay

Turns the displayed cursor on and off.

PdSetCursorPos

Sets the cursor position with coordinates specified as virtual screen coordinates.

PdSetCursorPosNSC

Sets the cursor position with coordinates specified as normalized screen coordinates.

Controls

PdSetControls

Allows an application to set various pointing device controls.

PdSetSystemControls

Allows an application to set various system-wide pointing device controls.

Transformations

PdTranslateVCToNSC

Translates virtual screen coordinates to normalized screen coordinates.

PdTranslateNSCToVC

Translates normalized screen coordinates to virtual screen coordinates.

System Programming in Multicontext Environment

PdAssignMouse

Assigns the mouse (or any pointing device) to the specified user number.

Performance Statistics Service

ClosePsSession

Is a synonym for PsCloseSession and is used for compatibility on older operating system versions.

DeinstPsServer

Is a synonym for PsDeinstServer and is used for compatibility on older operating system versions.

GetPsCounters

Is a synonym for PsGetCounters and is used for compatibility on older operating system versions.

OpenPsLogSession

Is a synonym for PsOpenLogSession and is used for compatibility on older operating system versions.

OpenPsStatSession

Is a synonym for PsOpenStatSession and is used for compatibility on older operating system versions.

PsCloseSession

Closes the session opened by the PsOpenStatSession or by the PsOpenLogSession operation.

PsDeinstServer

Deinstalls the Performance Statistics system service.

PsGetCounters

Returns the memory address of the structure containing the performance statistics counters.

PsOpenLogSession

Opens a log of the currently active processes in the ready queue or a log of the memory usage of short-lived and long-lived memory.

PsOpenStatSession

Opens a session for which BTOS performance statistics are collected by the Performance Statistics system service.

PoReadLog

Obtains the log for either the process run queue or the memory usage for the opened session.

PoResetCounters

Resets the performance statistics counters for the block-index specified.

ReadPoLog

Is a synonym for PoReadLog and is used for compatibility on older operating system versions.

ResetPoCounters

Is a synonym for PoResetCounters and is used for compatibility on older operating system versions.

Queue Manager

AddQueue

Activates a new queue.

AddQueueEntry

Adds an entry to the specified queue for processing by the appropriate queue server.

CleanQueue

Resets a queue to empty.

DeinstallQueueManager

Terminates operation of the Queue Manager and frees its memory partition.

EstablishQueueServer

Establishes a program as a queue server for the specified queue.

GetQmStatus

Interrogates the Queue Manager about usage statistics, as well as the queues of the specified type.

MarkKeyedQueueEntry

Locates the first unmarked entry in the specified queue with up to two key fields equal to the values specified, marks it as being in use, reads it into a buffer, and returns a queue entry handle for use in a subsequent RemoveMarkedQueueEntry operation.

MarkNextQueueEntry

Leads the first unmarked entry in the specified queue into a buffer, marks it as being in use, and returns a queue entry handle. Entries are marked in order of priority.

ReadKeyedQueueEntry

Obtains the first queue entry in the specified queue with up to two key fields equal to the values specified, reads it into a buffer, and returns the Queue Status Block.

ReadNextQueueEntry

Reads an entry from the specified queue into a buffer and returns the queue entry handle of the next queue entry.

RemoveKeyedQueueEntry

Locates an unmarked entry in the specified queue with up to two key fields equal to the values specified and removes it from the queue.

RemoveMarkedQueueEntry

Removes a previously marked entry from the specified queue.

RemoveQueue

Removes a queue dynamically.

RescheduleMarkedQueueEntry

Removes a previously marked entry from the specified queue or reschedules the entry if it is associated with a repeat time interval.

RewriteMarkedQueueEntry

Rewrites the specified marked queue entry with a new queue entry.

TerminateQueueServer

Notifies the Queue Manager that a queue server is no longer serving the specified queue.

UnmarkQueueEntry

Resets the specified queue entry as unmarked (not in use).

Sequential Access Service

Basic

SeqAccessClose

Releases a sequential access device from the exclusive access rights granted by a previous call to SeqAccessOpen.

SeqAccessControl

Specifies certain medium positioning operations that do not involve the transfer of user data to the medium. These operations include rewinding, unloading, retensioning, erasing, and writing file marks and may apply only to certain devices.

SeqAccessOpen

Provides exclusive access to the specified sequential access device and returns a sequential access handle to be used in subsequent operations.

SeqAccessRead

Reads data from the sequential access device and places it in a caller-specified buffer.

SeqAccessStatus

Returns the current status of the sequential access device and its medium. This operation also clears error condition flags.

SeqAccessWrite

Writes data to the medium mounted on a sequential access device. This operation also verifies the data transfer and returns a count of the amount of data unsuccessfully transferred.

Advanced***SeqAccessCheckpoint***

Causes the caller to wait until all of the data supplied by previous SeqAccessWrite calls has been successfully transferred to the medium, or until an exception condition occurs that prevents the data transfer.

SeqAccessDiscardBufferData

Allows the caller to discard buffered data from the output data stream. This operation is commonly used when an exception condition occurs (for example, end of medium, which signals that a new tape is to be mounted). It allows new data to be written onto the medium before resuming the stream of output data.

SeqAccessModeQuery

Returns information about the current operating characteristics of the sequential access device, such as whether the device is write-protected, whether it is operating in buffered mode, and what recording density is being used for the medium.

SeqAccessModeSet

Configures operating characteristics of the sequential access device, such as buffered or nonbuffered mode, medium transport speed, and medium recording density.

SeqAccessRecoverBufferData

Transfers data from the buffers of the Sequential Access Service to a caller-specified buffer. This operation is used when an exception condition occurs that leaves data in the Sequential Access Service buffers. In this case, an application program may wish to recover this data before resuming write operations to the medium.

Spooler Management

ConfigureSpooler

Sets or changes the spooler's configuration.

SpoolerPassword

Sends a file password to the spooler.

Voice/Data Services

Audio Service

AsGetVolume

Returns the volume level at which the Audio Processor plays back messages or emits system indicators (such as a beep on error) (SuperGen Series 5000 workstations only).

AsSetVolume

Sets the volume level at which the Audio Processor plays back messages or emits system indicators.

Telephone Service

TsConnect

Connects or disconnects the voice unit and telephone lines.

TsDataChangeParams

Changes the parity, line control, and other parameters to an open line.

TsDataCheckpoint

Checkpoints a data line.

TsDataCloseLine

Closes a data line and terminates the call.

TsDataOpenLine

Starts a data session using the modem.

TsDataRead

Moves a block of received data from the modem to the specified user memory.

TsDataRetrieveParams

Returns the parameters of the specified line.

TsDataUnacceptCall

Allows a program to receive a TsDataOpenLine request before the timeout expires.

TsDataWrite

Writes a data block to the modem.

TsDeinstall

Deinstalls the Telephone Service.

TsDial

Causes the dual-tone multifrequency (DTMF) encoder to generate the specified characters.

TsDoFunction

Causes the Telephone Service to perform functions similar to those available by pressing buttons on a typical two-line telephone.

TsGetStatus

Returns the state of the hardware and all users.

TsHold

Causes the specified line to be disconnected from all devices and placed on hold.

TsLoadCallProgressTones

Sets the call progress tones to be used for a subsequent TsDial request.

TsOffHook

Makes the specified line go off the hook.

TsOnHook

Causes the specified line to be disconnected from all devices and go onhook or “hung up.”

TsQueryConfigParams

Gets the Telephone Service configuration file name and the current configuration information.

TsReadTouchTone

Reads the DTMF signal from a telephone line or the Voice Processor™.

TsRing

Turns on or off the monitor ringing, allowing an application and user to select a ring frequency interactively.

TsSetConfigParams

Sets the Telephone Service configuration information.

TsVoiceConnect

Specifies the connection for a *TsVoicePlaybackFromFile* or *TsVoiceRecordToFile* operation when the flag *fAutoStart* in the Voice Control Structure is FALSE.

TsVoicePlaybackFromFile

Plays back voice data from the specified file.

TsVoiceRecordToFile

Copies voice data to the specified file.

TsVoiceStop

Causes the digitization to be suspended.

Section 36

Partitions and Partition Management

What is Partition Management?

This section describes partitions, the relationship between a partition and the program running in it, and the operations available to a program for managing other programs executing in partitions at the same time. Partitioned memory is supported for backwards compatibility with older operating systems. For a discussion of partitioned memory on multipartition, variable partition, and virtual memory operating systems, see “Programs and Run Files” in the section entitled “Introduction to Operating System Concepts.”

In “Program Management,” you were introduced to the operations available to a program for self-loading, self-exiting, and handling error conditions. Although there are some restrictions on operations that can be used when a program converts to a system service, the operations described in “Program Management” are used by any program executing in memory, regardless of whether the partition is managed by another program. For example, a partition-managed program can specify an exit run file to which it can chain. Furthermore, all programs must call an exit routine to terminate and should handle runtime error conditions. This section assumes you are familiar with the program management operations.

The operations described in this section are additional program management operations. They are typically used by a partition managing program such as Context Manager to create and manage partitions in a multiprogramming environment.

Partitions

Partitions can be different types and sizes, and can consist of different components.

Types

System memory consists of two basic types of partitions: application and system.

An *application partition* can contain an application program.

Application programs can use the keyboard and video display, and can allocate memory dynamically. If the program is an interactive command interpreter, such as the Executive, you can use the program to load other programs, such as the Editor, OFIS Document Designer, or Multiplan, into the partition.

A *system partition*, on the other hand, can contain the operating system or a dynamically installed system service. Because all programs (other than the operating system) start out as ordinary applications executing in application partitions, the emphasis in this section is on the application partition.

Fixed or Variable

An application partition can be fixed or variable. *Fixed* partitions always use a fixed amount of memory. *Variable* partitions, on the other hand, can grow or shrink with the needs of a program. The type of partition supported depends on the operating system memory management type. (For details, see “Memory Management Styles,” in the section entitled “Overview of Operating System Concepts.”)

A variable partition can use up to the maximum amount of memory specified at link time when the program to be loaded into it was sized. Furthermore, a program executing in a variable partition can share its code with another instance of the same program executing in a separate variable partition, provided the code and data in the program are separate.

Partition Components

A partition is not necessarily a “memory cell” into which a run file is loaded. Instead, a partition can be viewed as collection of one or more of the following components:

- Short-lived memory
- Common pool of unallocated memory
- Long-lived memory
- Program code
- User structure
- Local descriptor table (LDT) (protected mode only)

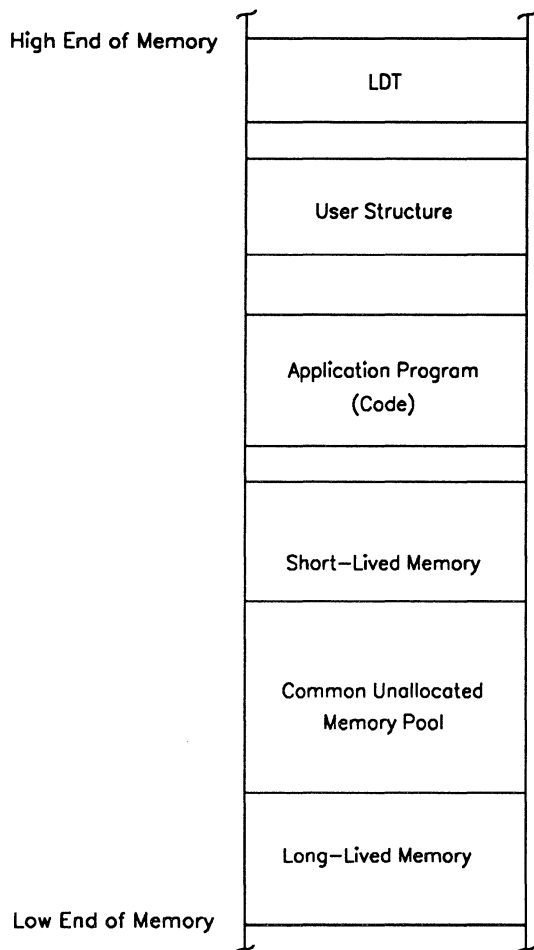
These components are first examined from the standpoint of how they may exist in memory. Following “In-Memory Relationships,” each component is described in detail.

In-Memory Relationships

The presence or absence of partition components and their locations in memory depend on various factors. In addition to the style of memory management, the way a program is linked can be a determining factor. Because the components of a partition can exist anywhere in the linear address space in protected mode, components are commonly referred to as *hypersegments*.

Figure 36-1 depicts the hypersegments as they may exist in linear memory in protected mode.

Figure 36-1. Hypersegments in Protected Mode



557.36-1

Long-lived memory, short-lived memory, and the unallocated memory pool are located contiguously.

On variable partition protected mode and virtual memory operating systems, all other hypersegments can be located anywhere in linear memory. On a multipartition real mode operating system, on the other hand, the entire partition is contiguous (and the user structure is typically located at the low end of memory). (See “Memory Management Styles,” in the section entitled “Overview of Operating System Concepts.”)

Each of the partition hypersegments is described in the paragraphs below.

Local Descriptor Table

The local descriptor table is a protected mode structure that contains segment descriptor entries for the program. All partitions loaded with protected mode programs that are not linked explicitly to use the global descriptor table have a local descriptor table component.

User Structure

The user structure is a loose collection of information that describes an application partition. A user structure does not become a part of a system partition when the program converts to a system service. (For details, see “Initialization and Conversion to a System Service” in the section entitled “System Services Management.”)

User Structure Contents

The user structure can contain the following partition structures:

- Partition configuration block
- Extended partition descriptor
- Application system control block (ASCB)
- Batch control block

The partition configuration block, extended partition descriptor, and ASCB are common to all application partitions. The batch control block is optional and supports the execution of Background Batch in the partition. It can be specified when the partition is created using the CreatePartition operation.

The section entitled “System Definitions” provides a brief description of each of these structures. For more comprehensive descriptions and the format of each structure, see the section entitled “System Structures,” in the *CTOS Procedural Interface Reference Manual*.

For application partitions that use the keyboard and screen, the user structure also can include the overhead necessary for the virtual character map and keyboard buffer. This overhead is specified by a partition managing program when it calls the CreatePartition operation. In addition, a program can call the ReservePartitionMemory operation to reserve an extended user structure. The extended structure is additional partition memory that can be used by the calling program.

User Structure and Multiprogramming

The user structure allows multiprogramming to take place. Say, for example, two programs were executing in different memory partitions. Only one of the programs could use the video hardware buffer to write to the real screen at any given time. If both programs called a VAM operation such as PutFrameCharsAndAttr or PosFrameCursor, the program not using the real screen can write the byte to an address in the virtual character map. Furthermore, if a partition is swapped to disk or extended memory, the current description of that partition is saved in the user structure so it can be restored when the partition is swapped back into memory again.

Partition managing programs as well as system services refer to the user structure of a partition to access information about that partition. Access is gained indirectly by calling the GetPStructure operation or by calling certain operations specifically for this purpose. Recommended methods of access are discussed in “Methods of Obtaining System Information” in the section entitled “System Definitions.” The operating system itself accesses these structures through the partition descriptor.

Program Code

Program code can be a separate hypersegment of a variable partition if the run file is linked with separate code and data (protected mode programs only). As such, the code can be shared by other instances of the same program executing in a different memory partition. Program code contains only processor instructions or read-only data and is never modified once it is loaded into memory.

Short-Lived and Long-Lived Memory

Short-lived and long-lived memory are allocated and deallocated dynamically through calls an application program makes to the operating system. (For details, see the section entitled “Memory Management.”)

All partitions contain short-lived memory. Unless a run file is explicitly linked with separate code and data and is loaded into a variable partition (protected mode programs only), both the code and static data are contained in the short-lived memory of the partition. A system service partition is created out of the short-lived memory of its partition. (See “Initialization and Conversion to a System Service” in the section entitled “System Services Management.”)

User Number

A *user number* is historically the same as a partition handle. It is a 16 bit identifier that uniquely identifies the program and/or the resources (such as file handles, short-lived memory, long-lived memory, and exchanges) associated with the partition. Because partitions are hypersegments that can be located anywhere in memory on protected mode operating systems, a user number is not associated with partition size or location.

Caution

No significance should be placed on the meanings of the bits in a user number. Programs should treat the user number as a handle only.

When a partition is created using `CreatePartition` or `CreateBigPartition`, the user number for the partition is returned. The user number can be used to refer to the partition in subsequent operations such as `GetPartitionStatus`, `LoadPrimaryTask`, and `RemovePartition`.

A previously assigned user number can be obtained by supplying the partition name to the `GetPartitionHandle` operation. The user number is subsequently used in calls such as `GetPartitionStatus` or `GetPartitionExchange`.

When a partition is removed using the `RemovePartition` operation, the specified user number is deallocated by the operating system and becomes available to be reissued.

A program can obtain the user number of its own partition by calling the `GetUserNumber` operation.

Partition Organization In System Memory

The discussion below describes partition organization in the linear address space on multipartition and variable partition operating systems. Linear memory organization on virtual memory operating systems differs somewhat. (See “Programs and Run Files” and “Memory Management Styles” in the section entitled “Overview of Operating System Concepts,” for details.)

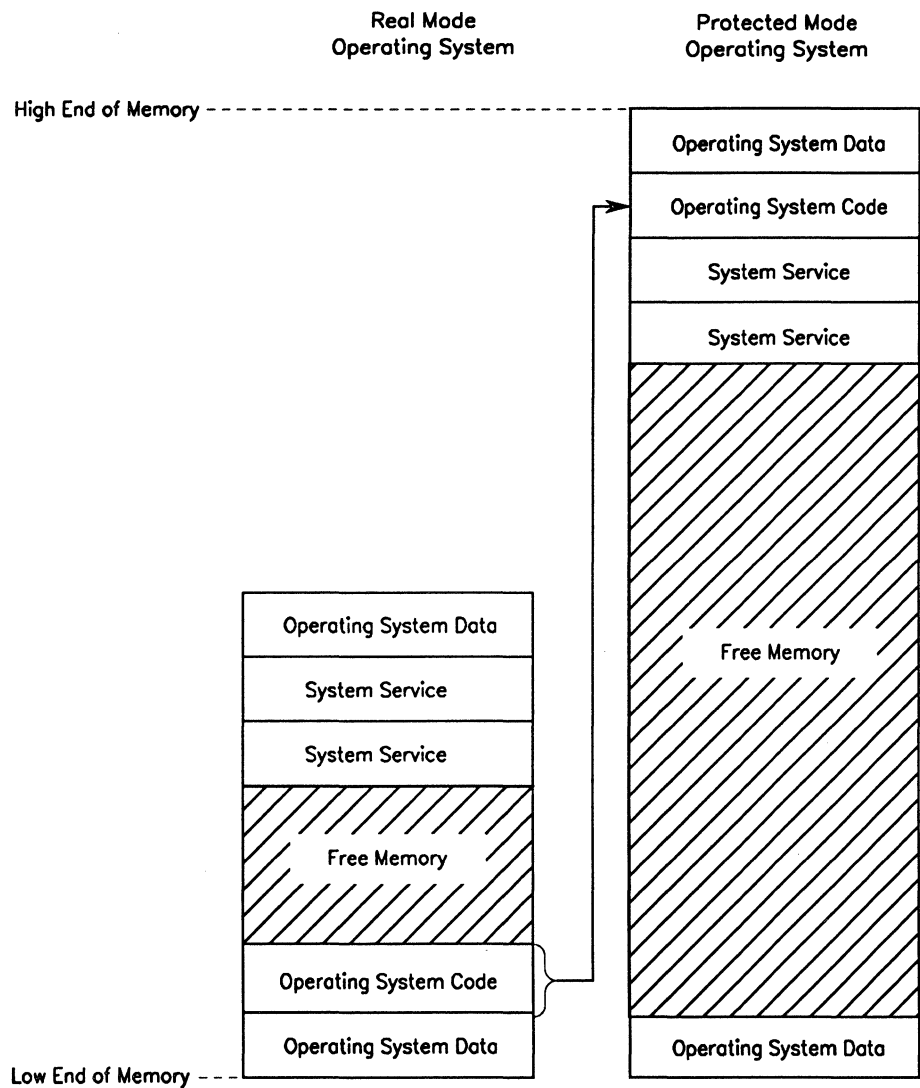
Partition organization in linear memory is depicted first at system initialization. Then it is shown after a number of partitions have been created under the direction of a partition managing program.

At System Initialization

Figure 36-2 shows how memory is organized at system initialization. When a system is initialized, the operating system is loaded into system partitions at the low address and high address ends of the linear address space. Dynamically installed system services are loaded into system partitions at the high address end. All remaining linear memory is *free memory*.

Programs executing in system partitions are system service programs. Such programs (other than the operating system) start as ordinary application programs and then use the ConvertToSys operation to change the status of their partition from application partition to system partition.

Figure 36-2. Memory Organization at System Initialization



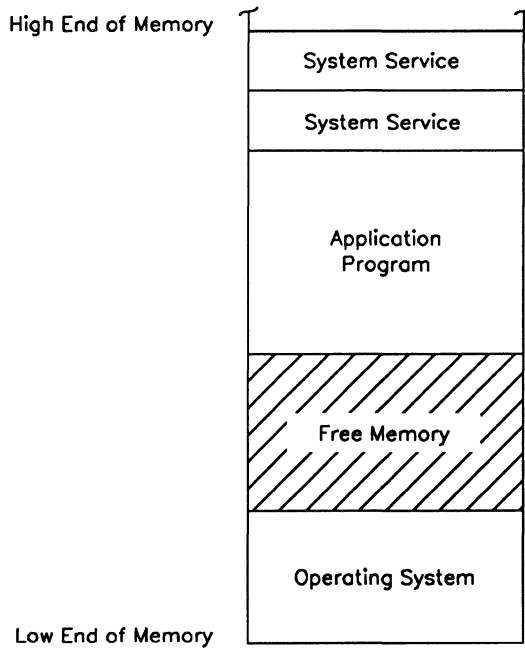
557.36-2

Single Application Partition In Memory

Figure 36-3 shows a single application partition called the *primary partition* containing a program in memory.

Compare the partition arrangement in this figure to that depicted in Figure 36-2 for a moment. On multipartition (real mode) operating systems, the application is loaded into the free memory just below the system services. Maximum addressability on multipartition systems is one megabyte. All executing applications and system services must fit in this one-megabyte space. On variable partition operating systems, a protected mode application is loaded below the system services but it can be located at a memory address extending beyond the first megabyte. Because real mode applications have only one megabyte of addressability, however, they can only be loaded into the first megabyte range.

Figure 36-3. Single Application Partition in Memory



557.36-3

Multiple Application Partitions In Memory

A partition managing program is designed to create and to manage other partitions, more than one of which can be in memory at once. Context Manager is such a program and is used in the following description of multiple application partitions in memory.

Note: *The memory management style depicted in this discussion of Context Manager is variable partition memory management.*

You must install all system services before installing Context Manager. You cannot, for example, use the Executive commands to install system services from an Executive program in a partition created by Context Manager.

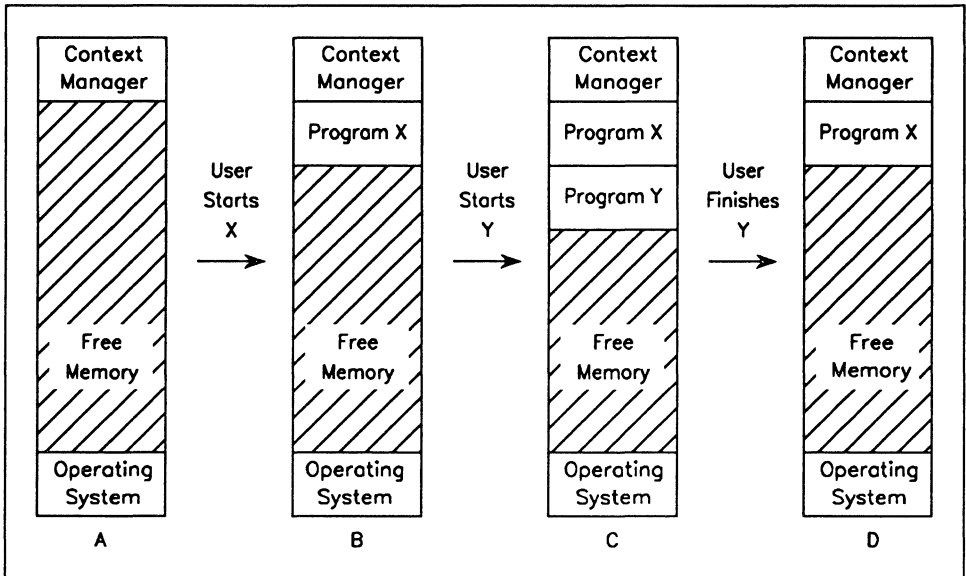
Figure 36-4 Part A shows what memory looks like when Context Manager is first loaded into the primary partition. Context Manager is at the high address end of free memory.

When the user selects an application to start, Context Manager dynamically creates a fixed or a variable application partition using the CreatePartition or CreateBigPartition operation. The new partition is created just beneath Context Manager, which remains in memory at the high address end. Context Manager then loads the selected application program into that partition using the LoadInteractiveTask operation. The remaining unused memory is free memory. (See Figure 36-4 Part B.)

Each additional program started from Context Manager is loaded just under the last until memory is full. Figure 36-4 Part C shows what memory looks like with the addition of a second program in memory.

When a user finishes a program, the partition that it was in becomes free memory as shown in Figure 36-4 Part D.

Figure 36-4. Multiple Application Partitions in Memory



557.36-4

Partition Swapping

When the user chooses to start a program from Context Manager and there is not enough free memory available to create a partition into which to load the program, the operating system selects which partition(s) to swap out to a file on disk or to extended memory. To do this, the operating system uses an algorithm, which takes the following into consideration:

- Whether the program is capable of swapping
- Whether the program is currently using the real screen and keyboard

When program(s) are swapped out of memory, the memory where the program(s) was located becomes free memory. This free memory is available

- To Context Manager for creating a new partition into which a new program can be loaded
- To the operating system for swapping a program back into memory from disk or extended memory

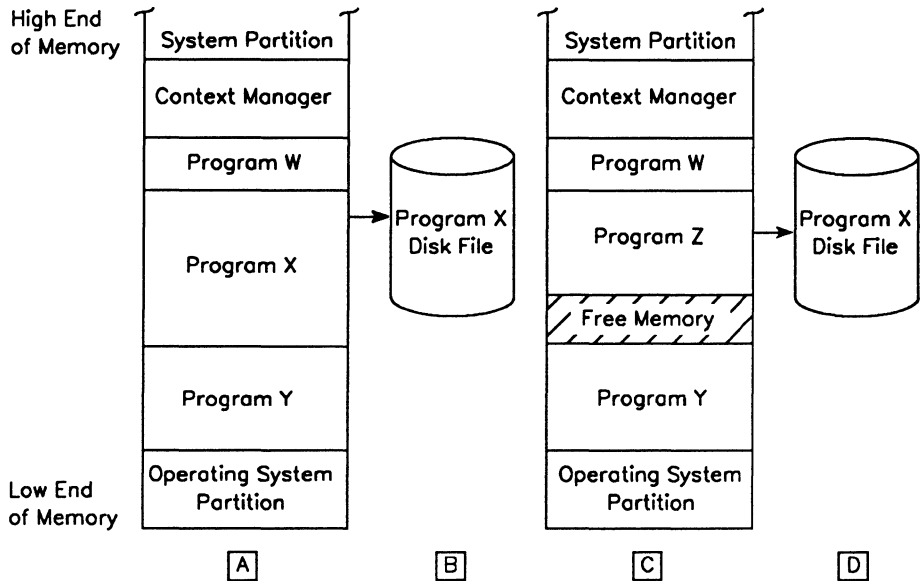
Figure 36-5 shows the following example sequence of what memory looks like when swapping occurs:

1. Figure 36-5 Part A shows Program W, Program X, and Program Y in memory partitions.
2. Figure 36-5 Part B shows Program Z swapped out to a disk file. The operating system selects to swap Program Z back into memory by swapping out Program X.
3. Figure 36-5 Part C shows memory with Program Z is swapped in. Note that the extra memory area taken up by the relatively larger Program X is free memory.
4. Figure 36-5 Part D shows Program X swapped to disk.

Note that Program Z's partition occupies the same memory area as Program X's partition did when it was in memory. Program X and Program Z have unique user numbers associated with their partitions.

You can create a swap file or use the operating system swap file by default. (For details, see the sections entitled "Configuring Workstation Operating Systems" and "Configuring Shared Resource Processor Operating Systems," in the *CTOS System Administration Guide*. Within each section, refer to the subsection called "Configurable Parameters," for swap file information.)

Figure 36-5. Swapping



557.36-5

Creating a Partition

Caution

It is not recommended that a program already under partition management use partition management operations to create and manage additional partitions. Assigning the keyboard and real screen using the AssignKbd and AssignVidOwner operations is done by an overseeing partition managing program. Use of these operations by programs already under partition management may create conflict.

To dynamically create a fixed or a variable application partition, a partition managing program uses the CreatePartition or the CreateBigPartition (protected mode) operation.

When `CreatePartition` or `CreateBigPartition` is called, the user number for the partition is returned. The partition managing program can use the user number to refer to the partition in subsequent operations such as `GetPartitionStatus`, `LoadPrimaryTask`, and `RemovePartition`.

Loading a Program

Once the partition is created, a partition managing program can load a program into it using the `LoadInteractiveTask` or `LoadPrimaryTask` operation. `LoadInteractiveTask` must be followed by a call to `AssignKbd`, and `LoadPrimaryTask`, by a call to `SwapInContext`.

Additional run files can also be loaded into the same partition in memory by the program management operation `LoadRunFile`. (For details, see “Application Partition with Multiple Run Files.”)

Terminating a Program

A partition managing program can use the `TerminatePartitionTasks` or `VacatePartition` operation to terminate an application program in another partition. Both operations terminate the program in the partition.

`TerminatePartitionTasks` differs in that it also loads and activates the partition’s exit run file, if one is specified (using the `SetExitRunFile` operation described in the section entitled “Program Management”). If no exit run file is specified, however, `TerminatePartitionTasks` and `VacatePartition` are equivalent.

When a program terminates, the operating system issues termination requests. *Termination requests* are messages that notify system services of a program’s termination. Upon receipt of a termination request, a system service releases resources, such as open files, that may be allocated to the terminating program. (For details on termination requests, see “System Requests” in the section entitled “System Services Management.”)

Removing a Partition

An existing vacant application partition can be removed using the `RemovePartition` operation. The specified user number is deallocated by the operating system and becomes available to be reissued in response to a call to `CreatePartition`.

An application partition is vacant when one of the following is true:

- It is first created.
- The current application program exits with no exit run file specified.
- The `VacatePartition` operation is performed.

Obtaining Partition Status

Any program (whether or not it is under partition management) can obtain status information about a specified application partition and the program executing in it (such as whether the program is sized) by using the `GetPartitionStatus` operation. `GetPartitionStatus` requires that the caller provide the user number of the partition. A previously assigned user number can be obtained by supplying the partition name to the `GetPartitionHandle` operation.

In addition, any program can access the individual partition structures in the user structure of another program by calling the `GetPStructure` operation with the appropriate structure code. (For details, see the description of `GetPStructure` in the *CTOS Procedural Interface Reference Manual*.)

Communicating Between Application Partitions

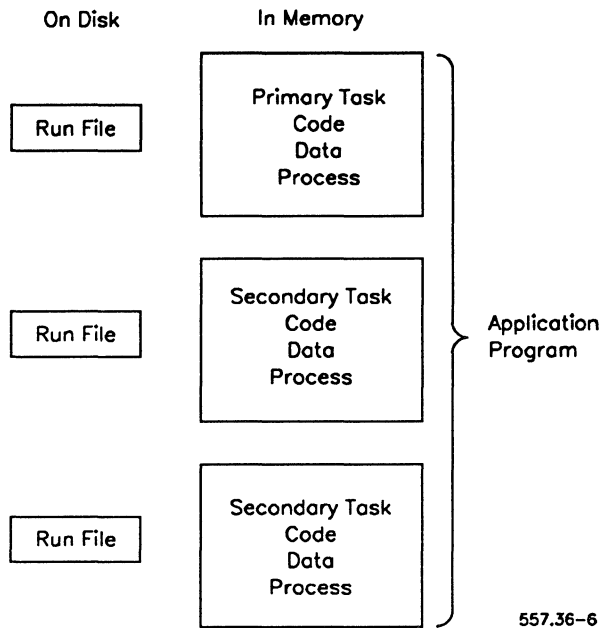
The Intercontext Message Service (ICMS) provides communication between application partitions managed by partition managing programs. (For details, see “Communication Between Application Partitions” in the section entitled “Interprocess Communication.” Also see your Context Manager manual.)

Application Partition With Multiple Run Files

Occasionally, an application partition will contain more than one run file. This occurs when the original program in a memory partition calls the LoadRunFile or LoadTask operation to load an additional run file into the same partition. If the LoadRunFile operation is used, the caller can conveniently remove the run file later using the DeallocRunFile operation. (For details, see the description of DeallocRunFile in the *CTOS Procedural Interface Reference Manual*.)

When there is more than one run file in a partition, the original program is the *primary task*. Any subsequent run files are *secondary tasks*. These tasks have a very special relationship in that they share the partition's system data structures and resources. Because these tasks are interwoven and function as a group, each is not a program, but a dependent part of the overall program in the partition. Figure 36-6 shows the relationships of these tasks to the program in a partition. In this manual *program* can mean one or more run files in a partition.

Figure 36-6. Multiple Run Files in an Application Partition



Partition Management Operations

The partition management operations described below are categorized according to use. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Basic Partition Management Operations

The following operations can be called by any program, whether or not the program is under the management of a partition managing program:

GetUserNumber

Allows a process to determine its own user number (which is historically the same as a partition handle).

GetPartitionHandle

Returns the user number of a specified partition. The requesting process must supply the name of the requested user number's partition as a parameter to this operation.

GetPartitionStatus

Returns status information about a specified application partition and the program currently executing in it.

SetPartitionName

Changes the name of the caller's partition. (Note that *SetPartitionName* *can* be used to change the name of any partition, but it is normally used to set the partition name of the caller.)

Swapping

The following operations can be called by any program, whether or not it is under the management of a partition managing program:

SetSwapDisable

Allows a program to specify that it can or cannot be swapped. Unlike SetPartitionSwapMode, SetSwapDisable is not long-lived (does not persist across a Chain operation).

SwapInContext

Requests that a specified user number's partition be swapped into memory. This operation is typically used by a partition managing program but can be used by any program provided the program that executes in the partition does not use the keyboard or screen.

GetPartitionSwapMode

Returns the swap mode established by the SetPartitionSwapMode operation.

SetPartitionSwapMode

Sets the swap mode for the specified partition. The swap mode is long-lived (persists across Chain operations).

LockInContext

Locks the specified partition in memory so that it cannot be swapped out. Specifying a user number of zero causes the caller's partition to be locked.

UnlockInContext

Removes the locked state previously assigned to the partition with the LockInContext operation. Specifying a user number of zero causes the caller's partition to be unlocked. When all locks on a partition have been removed using UnlockInContext, the partition can again be swapped.

Partition Creating Under Program Control

The following operations are typically used by a partition managing program that has control over use of the keyboard and screen:

CreatePartition

Creates a new application partition, assigns its name, and returns a user number.

CreateBigPartition

Is the same as *CreatePartition*, except that *CreateBigPartition* allows you to create a new application partition that is larger than 1 megabyte (protected mode only).

ReservePartitionMemory

Dynamically allocates additional memory in an application partition and returns a memory handle that uniquely identifies the memory. This operation is called by system services to reserve local partition memory for internal use.

AssignKbd

Assigns the keyboard to a partition.

AssignVidOwner

Assigns the screen to a partition.

CreateUser

Creates a variable partition, specifying the size of the partition system data structures.

Partition Loading Under Program Control

The following operations are typically used by a partition managing program to load programs into memory partitions, to terminate programs, and to vacate and remove partitions:

LoadPrimaryTask

Loads and activates the run file specified by the file specification in a vacant application partition.

LoadInteractiveTask

Is the same as `LoadPrimaryTask` but provides the additional option (by means of an *fDebug* parameter) to indicate whether or not the run file is to be debugged when it is loaded into the partition.

VacatePartition

Terminates the program in the application partition specified by the user number but does not load and activate the exit run file. The partition is left vacant.

RemovePartition

Removes the specified vacant application partition.

TerminatePartitionTasks

Terminates the program in the application partition specified by the user number and loads and activates the partition's exit run file.

SetPartitionLock

Declares whether a program executing in the specified application partition is locked. The locked partition cannot be vacated with the `VacatePartition` operation.

Loading Additional Tasks

The following operations are used to load additional run file(s) into a partition that contains an existing run file(s):

LoadRunFile

Allocates code or data and loads it into memory.

DeallocRunFile

Deallocates code or data allocated (and loaded) using the `LoadRunFile` operation and frees the selectors used to load the code or data.

LoadTask

Loads and activates an additional (secondary task) run file as part of the current program in the application partition.

Section 37

Timer Management

System Timers

The timer management facility provides for two types of system timers: a realtime clock (RTC) and a programmable interval timer (PIT).

The RTC has a message-based interface you can use for accurate timing over long periods.

The PIT has a pseudointerrupt interface you can use for timing short intervals.

Realtime Clock

The realtime clock (RTC) provides both the current date and time of day and the timing of *intervals* (in units of 100 milliseconds). (For a cluster workstation without a local file system, the current date and time are maintained at the server. For a cluster workstation with a local file system, the current date and time are maintained at both the server and at the cluster workstation.)

A client can request that a message be sent to a specified exchange either once after a specified interval or repetitively with a specified constant interval between sending operations. The first time a message is sent to an exchange can be up to 100 milliseconds earlier than specified. Subsequent intervals are timed exactly.

Programmable Interval Timer

The programmable interval timer (PIT) uses a 50 microsecond, high-resolution timing source. The PIT is controlled by a 16 bit counter and therefore has a maximum interval of approximately 3 seconds.

The PIT is used for high-resolution, low-overhead activation of user pseudointerrupt handlers. A client or an interrupt handler can request that a pseudointerrupt handler be activated after a specified interval.

Using the Timer Management Operations

A client can use timer management in the following ways:

1. By calling the `Delay`, `ShortDelay`, or the `Doze` operation.
2. By using the RTC service. The RTC service can be accessed by calling the `OpenRtClock` operation.
3. By using the PIT. The PIT can be accessed by calling the `SetTimerInt` and `ResetTimerInt` operations.

Using Delay and Doze

Both `Delay` and `Doze` allow the client process to suspend execution for a specified interval in 100 millisecond units. The difference between these operations is that `Doze` is only supported on protected mode operating systems and provides for more accurate timing.

If the client is swapped out while the `Delay` operation is being used, the client will remain swapped out until an event external to the client causes the client to be swapped back into memory. (Under Context Manager, for example, the user might switch contexts back to the client context by pressing the key combination `ACTION+NEXT`. This would cause Context Manager to call `SwapInContext` to swap the client back into memory again.) While the client is swapped out, its RTC timing service is suspended and does not resume until the client is swapped back into memory.

If the client is swapped out while the `Doze` operation is being used, the RTC service continues to work for it. As such, the client would be swapped back into memory automatically to resume execution when the specified time interval has elapsed.

Using ShortDelay

ShortDelay is an object module procedure in the standard operating system library. This procedure uses the PIT to time 1 millisecond intervals. Because of the overhead in setting up the timer, ShortDelay becomes less accurate if the interval to delay is very short.

Realtime Clock

The OpenRtClock operation initiates the use of a data structure provided by a client for control of complex RTC services. This data structure, the timer request block (TRB), is shared by the client and timer management. The CloseRtClock operation terminates sharing of the TRB.

The TRB defines the *interval* after which a message is sent to a specified exchange. The message can be sent either once after the specified interval or repetitively with the specified constant interval between send operations. The message is actually the memory address of the TRB.

The client must acknowledge receipt of the TRB (as described below) before timer management will send the same TRB again. This ensures that system resources (link blocks) are not consumed by queuing the same TRB at the same exchange many times. The client can also dynamically modify other fields of the TRB.

(For the TRB format, see “Timer Request Block Format” in Section 3, “System Structures,” in the *CTOS Procedural Interface Reference Manual*.)

Every 100 milliseconds, the timer management RTC interrupt handler performs the following sequence of operations on each active TRB. This sequence ensures that timer management will not send the same TRB again until the client decrements the *cEvents* field to 0.

1. If the counter field is 0, do nothing.
2. Decrement the counter field by 1.
3. If the counter field has not become 0, do nothing more.
4. If the *cEvents* field is 0, send a message to the exchange specified by the *exchResp* field. The message is the memory address of the TRB (not a copy of the TRB).
5. Increment the *cEvents* field by 1.
6. Copy the *counterReload* field to the counter field.

Timing a Single Interval

A client should use the sequence below to initialize a TRB to time a single interval.

1. Set the counter field to 0.
2. Call the *OpenRtClock* operation.
3. Set the *cEvents* field to 0.
4. Set the *counterReload* field to 0.
5. Set the counter field to the chosen interval.

Use the *Wait Check Kernel* primitive (specifying the exchange specified by the *exchResp* field) to receive the indication that the interval expired. Remember that the RTC only operates in units of 100 milliseconds. Thus, if the counter field is set to 3, the TRB can be sent to the *exchResp* exchange in as few as 200 milliseconds or as many as 300 milliseconds. To reuse the TRB to time another single interval, repeat the sequence above from step 3.

Repetitive Timing

A client should use the sequence below to initialize a TRB for repetitive timing.

1. Set the counter field to 0.
2. Call the `OpenRtClock` operation.
3. Set the *cEvents* field to 0.
4. Set the *counterReload* field to the chosen interval.
5. Set the counter field to the chosen interval.

The first time that the TRB is sent to the *exchResp* exchange can be up to 100 milliseconds earlier than specified. Subsequent intervals are timed exactly. Exact timing is guaranteed because the counter field of the TRB is decremented even if the client has not finished processing the previous event. The *cEvents* field provides a continuous count of the events that have occurred but are not yet processed. If the client is too slow, the count in the *cEvents* field becomes ever larger. Under these circumstances, the count in the *cEvents* field provides a measure of how far behind processing has fallen.

The client should use the sequence below to process the TRB. This sequence avoids a race condition and yet processes the correct number of events.

1. Receive indication that the interval expired by using either the `Wait` or `Check` primitive and specifying the exchange specified by the *exchResp* field.
2. If the *cEvents* field is 0, processing is complete; return to step 1. (In this sequence, it is possible to receive a TRB in which *cEvents* is 0; thus it is necessary to perform this test before processing the event.)
3. Process the event. Processing is application-specific.
4. Decrement the *cEvents* field by 1. (It is not necessary to decrement the *cEvents* field in a single instruction unless the client is keeping a count of events.)
5. Repeat the processing sequence from step 2.

Programmable Interval Timer

Your application can access the high-resolution programmable interval timer (PIT) with the `SetTimerInt` and `ResetTimerInt` operations.

`SetTimerInt` starts the PIT. It specifies the memory address of a timer pseudointerrupt block (TPIB) in local memory that must be allocated by the caller. When the specified time interval (in units of 50 microseconds) expires, `SetTimerInt` performs either of two functions. It can

- Call a pseudointerrupt handler in the client to receive a pseudointerrupt. This option causes the client program to be locked into memory.
- Send a message to an exchange. The message is the memory address of the TPIB. This option avoids locking the client program into memory.

The exchange option can save on the amount of system memory being used. When an interrupt handler is established, the operating system automatically locks the entire partition using it into memory. (On demand paged, virtual memory systems, the paging service locks in the partition.) If your application uses a lot of memory and the interrupt handler uses very little, all the system memory taken up by your application partition is tied up. Very often the only function the pseudointerrupt handler performs is to send a message to an exchange. In this situation, the exchange option circumvents execution of the interrupt handler entirely and avoids consuming excessive amounts of system memory.

You must specify which option you want `SetTimerInt` to perform through the `pIntHandler` field in the TPIB structure. (For details on the format of the TPIB, see “Timer Pseudointerrupt Block” in Section 3, “System Structures,” in the *CTOS Procedural Interface Reference Manual*.)

Note: *Other interrupt activity may result in a slightly longer PIT timed interval than requested. Very short requested intervals are particularly susceptible to this effect and can cause significant system overhead.*

It is sometimes convenient to have a single pseudointerrupt handler service the pseudointerrupts associated with multiple TPIBs. To do this, the *pRqBlkRet* field of each TPIB must be the memory address of the same 4 byte memory area (or *pRqBlkRet* can be 0), and the *SetTimerInt* operation must be invoked for each TPIB. The pseudointerrupt handler must examine this 4 byte memory area to determine which TPIB caused activation of the pseudointerrupt handler. Even when the pseudointerrupt handler is serving only a single TPIB, *pRqBlkRet* must still be the memory address of the otherwise unused 4 byte memory area (or *pRqBlkRet* can be 0).

The *ResetTimerInt* operation terminates a previous *SetTimerInt* operation.

Timer Management Operations

The timer management operations are described below. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Delay

Delay

Delays the execution of the client for the specified interval. Timing is suspended while a client is swapped out of memory.

Doze

Delays the execution of the client for the specified interval. Timing continues if the client is swapped out of memory.

ShortDelay

Delays the execution of the client for the specified interval. The interval is shorter than that used in the Delay or Doze operation (1 millisecond rather than 100 milliseconds).

Realtime Clock

OpenRtClock

Establishes a TRB between the client and timer management.

CloseRtClock

Terminates the use of the specified TRB.

Programmable Interval Timer

SetTimerInt

Establishes a PIT pseudointerrupt handler.

ResetTimerInt

Terminates the TPIB initiated by a SetTimerInt call.

Section 38

Virtual Code Management

What is Virtual Code Management?

The virtual code management facility (commonly known as the “swapper”) allows you to run a program that is larger than the available memory in an application partition. To achieve this, only portions of the code exist in memory at any given time; the remainder is on disk. It is the function of virtual code management to ensure that the portions of the code that are currently needed for execution are actually in memory.

On multipartition and variable partition operating systems, the virtual code management facility is required to run applications larger than available memory. On virtual memory operating systems, the virtual code management facility is not needed. Applications that use virtual code management, however, can be run without change.

The virtual code management facility is a set of object module procedures in the standard operating system libraries. These procedures are linked with the program and become part of the run file. In protected mode on variable partition operating systems, a portion of the virtual code management facility resides in the operating system.

This section presents the virtual code management facility from a theoretical point of view. It describes how the operating system handles the movement of program segments between disk and memory. For practical guidelines on how to incorporate the virtual code management facility into your programs, see *CTOS/Open Programming Practices and Standards*.

Overlays

Each application program using the virtual code management facility is divided into variable-length code segments. The segments contain one or more complete procedures in object modules. One or more code segments are resident in memory. The others reside on disk in the run file. The virtual code management facility brings them into memory as they are needed.

A code segment in memory that is no longer needed is discarded, and another code segment (called an *overlay*) is read into its place in memory. When the first code segment is needed again, it is reread from the run file. Under this system, only code segments, and not data segments, are read into memory and discarded as necessary. Nothing is written back to disk, so there is no need for a disk swap file.

On virtual memory operating systems, overlays are simply treated like ordinary program pages: they are faulted into physical memory frames by the paging service when the code they contain is requested.

Note: *Programs that use overlays can be run without change on a virtual memory operating system. The overlays are simply treated like ordinary program pages: they are faulted into physical memory frames by the paging service when the code they contain is requested.*

Writing or Retrofitting Overlay Programs

You can write a program with the intention of using the virtual code management facility, or you can rather easily retrofit an existing program to use it. Few, if any, source program changes are needed: using the virtual code management facility mainly involves specifying to the Linker your desired grouping of object modules into code segments. You are responsible for dividing your program into blocks of code that can be swapped and for deciding how much memory to allocate for the swap area.

(See the section on using virtual code management in *CTOS/Open Programming Practices and Standards* for an overview of how to specify the modules you want to place in overlays. For additional information on the Linker, see the *CTOS Programming Utilities Reference Manual: Building Applications*.)

Model Overview

The virtual code management facility allows the execution of programs whose code size exceeds the size of the partition in which they are run. To achieve this, only portions of the code exist in memory at any given time; the remainder are on disk. It is the job of the virtual code management facility to ensure that the portions of the code that are currently needed for execution are actually in memory.

The code in the run file of a program using the virtual code management facility either is part of one of several *overlays*, or is resident. (Hereafter, a program that uses the virtual code management facility is called an *overlay program*.) When the overlay program begins execution, the resident code is loaded into memory, where it remains for the duration of the program's execution. At some point in the program's execution, when a call is made to a procedure in one of the overlays, the virtual code management facility reads that overlay into memory into an area of memory called the *overlay zone* so that the program can continue execution.

The virtual code management facility keeps as many overlays as possible in memory at once. When another overlay that would exceed the available space is called into memory, the virtual code management facility uses a least-recently-used (LRU) algorithm to determine which currently resident overlay to discard.

The virtual code management facility is designed to run in both real mode and protected mode and participates with the compatible run file format. That is, if an application program is written following the rules for protected mode programs, a single overlay program run file can be created that will run in both real mode and protected mode. Which mode it actually runs in depends on which operating system is present.

Note: *On virtual memory protected mode operating systems, overlays are treated like ordinary program pages: they are resident in physical memory frames when in use.*

The virtual code management facility operates quite differently in protected mode than in real mode.

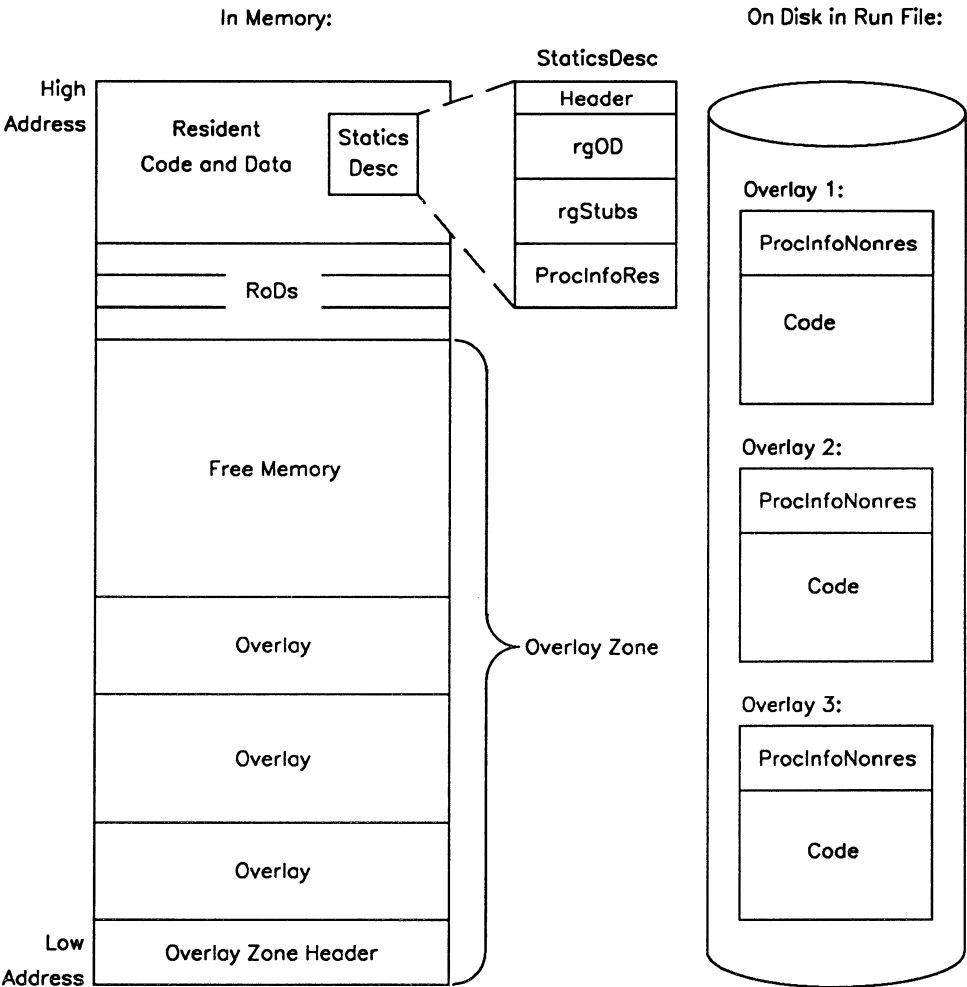
Data Structures

The virtual code management facility uses several data structures to keep track of the current locations of all of an overlay program's procedures (in memory or on disk, and in what overlay).

When an overlay program is linked, the Linker builds several data structures within it for use by the virtual code management facility. When the program is running, the arrangement of its parts is as shown in Figure 38-1. The program's resident code and data are in high memory.

To show all aspects of the arrangement, the figure depicts the memory layout at some point during program execution, after several overlays have been brought into memory and discarded.

Figure 38-1. Virtual Code Facility Data Structures



557.38-1

Overlay Zone Header

The *overlay zone header* is at the low end of the overlay zone. This structure describes the overlay zone, indicating how much space is used by overlays and (for real mode only) how much by return overlay descriptors (RODs). (For details on using RODs, see “Intercepting Returns.”) It also contains other reference information, including the locations of the StaticsDesc data structure and some of its substructures (described below).

StaticsDesc

The *staticsDesc* structure is in the data segment (DGroup) of the overlay program. It consists of the following:

- A self-descriptive header
- An array of overlay descriptors (rgOD)
- An array of stubs (rgStubs)
- A ProcInfoRes structure

The overlay descriptors array (rgOD) contains an entry for each overlay in the program, indexed by overlay number. Each overlay descriptor identifies the location and size of the overlay in the run file.

The stubs array (rgStubs) contains a stub for each program procedure. In protected mode, the procedure's stub contains the protected mode selector and the offset of the procedure. In real mode, the stub for a procedure contains either the address of the procedure's current address in memory or, if the stub's procedure is not resident in memory, the address of the OverlayFault procedure.

The ProcInfoRes structure describes those procedures that are in the permanently resident portion of program code. Its header tells how many procedures are present in the resident code segments and identifies the index of the stub corresponding to the first public procedure in the resident. A *public* procedure is a procedure that can be accessed by other modules.

Return Overlay Descriptors

The return overlay descriptors (RODs) are overlay identifiers used by the virtual code management facility in real mode when a return is done to a procedure that was discarded after it issued the corresponding call. (For details, see “Intercepting Returns.”) RODs are not used in protected mode.

ProcInfoNonres

All code segments in overlays reside in the overlay program’s run file on disk. The ProcInfoNonres structure is at the head of each overlay code segment. It contains the index of the corresponding overlay descriptor (for example, what overlay this is) and its size. It also contains a time-stamp field for use with the LRU algorithm.

Note: *For overlays to work correctly in real mode, the contents of ProcInfoNonres must be in sorted order.*

Like the ProcInfoRes structure, ProcInfoNonres identifies the index in the stubs array of the stub corresponding to the first procedure in this overlay that can be accessed by other modules. Additionally, it tells the number of procedures in the overlay. Finally, it identifies these procedures as near or far:

- A *near* procedure is referenced by the offset (IP) of the procedure’s memory address. Near procedures can be called only by other procedures within the same module.
- A *far* procedure is referenced by both its code segment (CS) and its offset (IP). Far procedures can be called by procedures within the same or from within a different module.

(For details on how the virtual code management facility handles these procedures, see “Intercepting Returns.”)

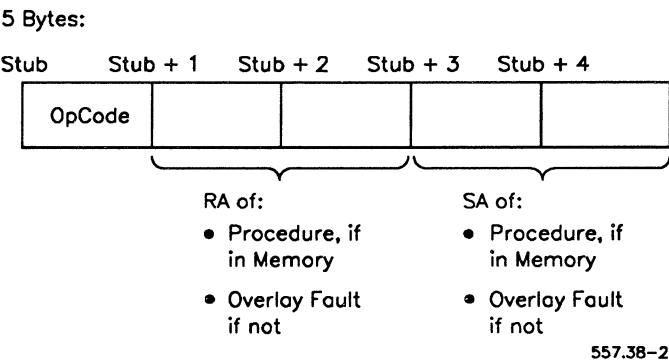
In real mode, the virtual code management facility needs the information provided by the ProcInfoNonRes structure when it traces the stack to discard an overlay.

The stubs array contains one stub for each program procedure. The Linker changes all program procedural calls from Call Direct to Call Indirect as follows:

```
CALL DWORD PTR [stub + 1]
```

Thus, each procedure is called through its corresponding stub. A stub has the 5 byte structure shown in Figure 38-2. In real mode, the first byte is either a JMP or a CALL instruction (opcode); in protected mode, the first byte always is a JMP. The remaining 4 bytes (in either mode) are a procedural address.

Figure 38-2. Stub Structure



Protected Mode Operation

In protected mode on variable partition operating systems, because each overlay is a separate segment, each overlay has a unique descriptor in the local descriptor table (LDT). (For details on protected mode structures, see the Intel Manuals listed in “Related Documentation,” at the beginning of this manual.) The present bit within the descriptor indicates whether or not the segment is in memory. When an overlay program is first loaded into memory, the operating system marks the descriptors for all the overlays as *not present*.

Whenever any of the discarded overlays are referenced (whether a procedure is being called or being returned to), a segment not-present fault will occur. A *segment not-present fault* is an interrupt from which control is passed to the segment not-present fault interrupt handler. The segment not-present fault interrupt handler is the part of the virtual code management facility that resides in the operating system.

The segment not-present handler must determine which overlay is needed before it can read it into memory. The processor supplies to the handler the selector that caused the fault. The virtual code management facility knows the selector of the first overlay in the LDT. It, therefore, can determine the overlay number for the desired overlay. It then uses the overlay number to index into the overlay descriptors array to find the address of the overlay on the disk. Virtual code management then performs the following functions:

1. It makes room in the overlay zone.
2. It reads in the overlay.
3. It updates the descriptor to reflect the overlay address.
4. It sets the descriptor present bit.
5. It restarts the instruction.

Real Mode Operation

Because real mode does not employ LDTs, overlay operation is somewhat different. These differences are described in the paragraphs that follow.

Intercepting Calls

In real mode, the stub contains the address of the `OverlayFault` procedure for each nonresident procedure (which, when the program is loaded, includes all procedures in overlays).

When a nonresident procedure is called, the call goes indirectly by means of the stub to `OverlayFault`. The `OverlayFault` procedure

- Determines which overlay it should bring into memory by analyzing its own address, which is constructed using a flexible additive address mechanism
- Examines the last 2 bytes of the original Call Indirect instruction to determine which stub the call came through, and therefore which procedure within the overlay is desired

Intercepting Returns

Note: *The following discussion assumes knowledge of stack format. (See the section entitled “Stack Format and Calling Conventions” in the CTOS/Open Programming Practices and Standards Guide for details.)*

In real mode, the virtual code management facility also intercepts returns to calling procedures. A calling procedure may be discarded from memory before it receives a return. A fatal error would occur if a return were made to a memory location previously occupied by a procedure that had since been discarded.

When the virtual code management facility has chosen an overlay to discard, it performs the following procedures:

1. It searches back through the stack to find the return address of the procedure being discarded.
2. It overwrites the return address with the OverlayReturnFault procedure's address.

This *stack trace* (exemplified below) is possible because the current stack base pointer (BP) is the memory address of the stack containing the BP address of the previous frame. (A *stack frame* is all of the information that is pushed on the stack when a procedure is called. The frame includes the parameters passed to the procedure and the information the procedure pushes on the stack during the course of execution.)

The BP of the previous frame, in turn, contains the previous BP, and so on, in a chain. The virtual code management facility follows the chain of BPs, checking the return addresses as it goes, and overwriting any in the discarded procedure with the OverlayReturnFault address.

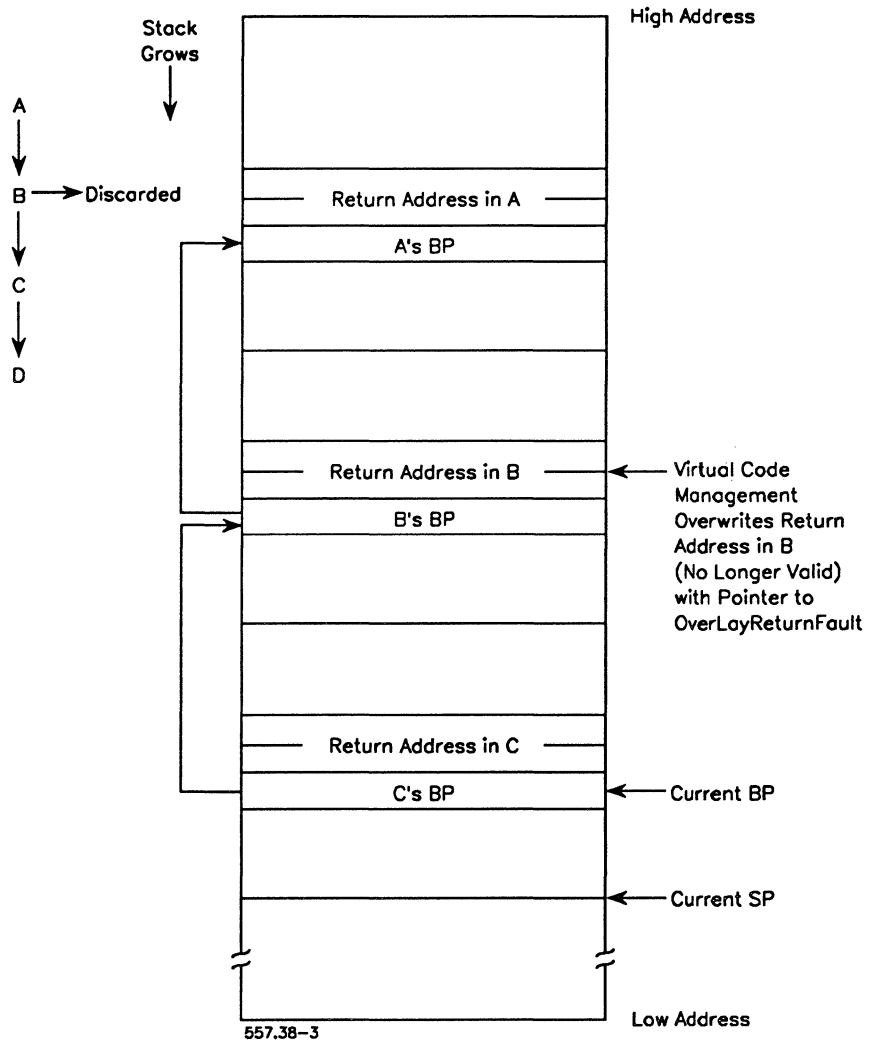
At this time, a return overlay descriptor (ROD) also is created for the discarded overlay, if the discarded overlay has any returns outstanding.

When the return occurs, it goes to OverlayReturnFault, the address of which now appears as the return address on the stack. The ROD identifies the overlay needed and the procedure within that overlay. OverlayReturnFault then brings this overlay into memory and passes control to the procedure, thus completing the call/return cycle.

OverlayReturnFault now marks the ROD as free so that RODs do not accumulate. (The number of existing RODs at any given time always equals the number of nonresident procedures with outstanding calls.)

Figure 38-3 illustrates stack tracing when an overlay is discarded.

Figure 38-3. Tracing the Stack



The scenerio leading up to Figure 38-3 is described as follows.

When Procedure A (in the resident portion of code) calls Procedure B (in an overlay), the overlay manager brings B into memory. B, in turn, calls Procedure C (also in an overlay), and C is brought into memory. Now Procedure C attempts to call Procedure D, but there is not enough room for D's overlay in the overlay zone.

The virtual code management facility examines its statistics and concludes that Procedure B is in the LRU overlay and should therefore be discarded. (Procedure B still expects a return from Procedure C.) The virtual code management facility discards the overlay containing B, creating a ROD to identify B. During this process, the virtual code management facility must trace the stack to overwrite the return address of B with the OverlayReturnFault address.

Figure 38-3 shows the stack format at this point. By convention, the BP register contains the addresses of variables that are local to the current procedure. It is therefore necessary that the program save the BP of the calling procedure so that it can be restored at the return. The positions of the BPs and the return addresses of the procedures are shown in Figure 38-3. Each BP contains the address of the previous BP. The virtual code management facility can jump from BP to BP, examining each return address and overwriting any return address belonging to the overlay that is being discarded.

When the virtual code management facility reaches a BP containing an address that matches the saved address of the initial stack pointer (SP), it terminates the trace.

Figure 38-3 is a simplification showing only far calls and returns. The structures, ProcInfoRes and ProcInfoNonRes, identify each procedure within their overlays as near or far. The virtual code management facility refers to these structures to determine whether it should read both a CS and an IP as a procedure's address (for a far procedure) or only an IP (for a near procedure).

After the overlay has been discarded, the virtual code management facility compresses the remaining overlays toward the low end of the zone and brings in D.

Procedure D now executes and then returns to C, which is straightforward because C is still in memory. C, however, returns to the address on the stack where B's return address normally would be, but the return now goes to OverlayReturnFault.

OverlayReturnFault analyzes this return address, accesses the correct ROD, and determines that the overlay that contains Procedure B must be brought back into memory. It then swaps B in, discarding another overlay if necessary. (Note that it is perfectly acceptable to discard the returning procedure to bring in the procedure receiving the return.)

Importance of Call/Return Conventions

Because of the stack-tracing scheme employed in real mode operation, your program must adhere to accepted call/return conventions. If the stack format is not what the tracing algorithm expects, the overlay program fails during the process of discarding an overlay. Note that the virtual code management facility does no stack tracing in protected mode. Nevertheless, you should follow these conventions to create a compatible run file (that is, a run file that allows your program to operate correctly in protected mode and real mode).

Calls to Procedural Addresses

In some real or protected mode programs, it is necessary to call a variable that is a procedure address rather than the actual procedure. The actual procedure to be used may be determined only at run time.

The virtual code management facility can handle such calls as well as standard procedure calls. The first byte of the stub is either a JMP or a CALL instruction.

In an overlay program, the Linker assigns the address of the stub's first byte to all memory locations within a program that contains references to the procedure. If the procedure to be called is

- Resident in memory, this byte is the JMP instruction, and the remaining 4 bytes are the address to which the jump should occur.
- Not resident in memory, there are two cases. For protected mode, the stub's first byte is the JMP instruction, and the remaining 4 bytes are the address of the procedure. The referenced descriptor is marked "not present." For real mode, the stub's first byte is the CALL instruction, and the remaining 4 bytes are the address of the OverlayFault procedure, which in turn brings the needed overlay into memory.

OverlayFault knows the stub from which a real mode nonresident procedure was called and thus can determine what procedure is needed.

Adjusting Addresses

In real mode, once an overlay has been brought into memory, the overlay manager overwrites the stub address of a frequently called procedure with that procedure's actual current address in memory. Thereafter, performance is improved as calls to that procedure go to it directly until that overlay is discarded.

The overlay manager keeps track of calls to an overlay while the overlay is in memory. By doing this, the overlay manager can determine the most active overlays, which are retained in memory.

In real mode, however, once a procedure stub address has been overwritten with the procedure's actual memory address, calls to the procedure no longer go through OverlayFault and are not logged. To compensate for this omission, your program can call the MakeRecentlyUsed operation, which prevents an overlay from being inadvertently discarded from memory. This operation is unnecessary in protected mode: if it is called, it will perform no function other than to return status code 0 ("ercOK").

When several overlays are in memory and the overlay manager needs to bring in another one for which there is not enough room, it uses this call-frequency data with its LRU algorithm to choose an overlay to swap out.

To enable reinitialization of its frequency-of-use log and to determine the new pattern of overlay use, the following compression procedure is performed:

- All remaining overlays are compressed toward low addresses.
- For real mode, the actual addresses of procedures within the overlays change. For protected, the descriptors for each moved overlay are updated to reflect their new locations.
- For real mode, all stubs are readjusted to the OverlayFault's address.

After this compression, the new overlay is brought into memory just above the highest existing overlay.

Overlays are available to programs that consist of more than one run file in an application partition. (For details, see “Application Partition With Multiple Run Files” in the section entitled “Partitions and Partition Management.”) The first run file contains the *primary task* and is loaded by the Chain, ErrorExit, Exit, LoadInteractiveTask, or the LoadPrimaryTask operation. A subsequent run file loaded into the same partition contains a *secondary task* and is loaded by the LoadTask operation. A secondary task, however, cannot be virtual if the primary task already uses virtual cCode management.

Virtual Code Management Operations

The virtual code management operations described below are categorized as basic or advanced. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Basic

InitOverlays

Initializes the virtual code management facility.

InitLargeOverlays

Initializes the virtual code management facility for large overlays.

Advanced

GetOvlyStats

Returns the size of the largest overlay, the size of the second largest overlay, and the total size of all overlays.

GetCParasOvlyZone

Returns the size of the overlay buffer measured in paragraphs.

ReinitOverlays

Allows the user to change the size of the overlay buffer to recover memory or extend the overlay buffer for better performance.

ReinitLargeOverlays

Is identical to *ReinitOverlays*, except the user describes the length of the overlay buffer as a count of paragraphs instead of bytes.

MoveOverlays

Changes the location of the overlay zone.

MakePermanent

Makes the overlay permanently resident in memory until it is released with a call to *ReleasePermanence*.

MakePermanentP

Makes an arbitrary overlay permanently resident in memory until it is released with a call to ReleasePermanence.

ReleasePermanence

Releases all overlays from permanent residence in memory.

MapIOvlyCs

Takes an overlay index and returns the address in memory of where the overlay is currently located.

MapCsIOvly

Takes the CS part of a memory address and returns the overlay in which that address is currently contained.

MapPStubPProc

Returns the last 4 bytes of a stub, which contain the address of a procedure.

MakeRecentlyUsed

Prevents an overlay from being inadvertently swapped out.

UpdateOverlayLru

Is called from within one overlay to prevent any other overlay from being swapped out by updating the time of its most recent use so that it appears to have 0 age.

EnableSwapperOptions

Allows an arbitrary operation to be called each time OverlayFault is called. This call works in real mode only.

DeallocateRods

Removes outstanding RODs when the stack is unwound in an assembly language program.

ReinitStubs

Sets all stubs, as a one-time reset, to contain the OverlayFault address.

Section 39

Dynamic Link Libraries

What are Dynamic Link Libraries (DLLs)?

This section describes dynamic link libraries (DLLs), guidelines on how to write them, and how applications can use them. The section compares DLLs to some of the other more familiar CTOS services. It also points out important caveats you should be aware of when writing DLLs.

A DLL is a loadable file, consisting of one or more procedures a program can call. The calling program or *client* of a DLL can be another DLL. DLLs can be dynamically linked to clients at program load or at runtime.

Under loadtime dynamic linking, the operating system automatically loads DLLs needed by a client when the client is loaded. Runtime dynamic linking requires that the client explicitly ask the operating system to load DLLs and to obtain the addresses of the DLL procedures it wants to use dynamically. Unless runtime linking is specifically mentioned, this section describes loadtime linking because it is more commonly used. Dynamic linking (also known as *late binding*) is supported on virtual memory operating systems.

DLLs relieve the system of storing multiple copies of library procedures. Once a DLL is loaded, its code in memory is shared among all client programs, not just among instances of the same program. Data segments in DLLs may be shared among all the DLL's users or may be unique for each user.

Dynamic Link Library Terminology

Terms relating to dynamic link libraries are described in Table 39-1. You may need to refer to this table from time to time as you read about DLLs.

Table 39-1. DLL Terms

Term	Definition
Address mapping	Converting linktime addresses to runtime addresses.
Dynamic linking	Modifying references to the DLL to point to the DLL as it exists in memory.
Export	A procedure in a DLL, or exporter, that may be directly accessed by a client.
Global initialization	DLL initialization procedures called once when the first DLL client is loaded.
Global segment	A DLL segment, a single copy of which is shared by all clients.
Import	A DLL procedure called in a client program, or importer.
Import library	A library containing the names of DLLs and DLL procedures to be called by a client. The Linker uses the client's import library to resolve external references to DLL procedures in the client program.
Instance initialization	DLL initialization procedures called each time a new client first references the DLL.
Instance segment	A DLL segment, an individual copy of which is made for each client.
Module definition file	A file that defines the specific requirements of a DLL or DLL client module to the Linker. A module definition file, for example, describes segment attributes such as whether a segment is a global or an instance segment.

DLLs Compared to Other CTOS Services

In the paragraphs below, DLLs are compared to other services available to an application executing on a CTOS operating system. These services are

- Object module libraries statically linked to applications
- System-common services
- Request-based system services

Table 39-2 summarizes key information about each of the service types. Note that a DLL can be a client of another DLL, which can be the client of a third DLL and so on. Calling and executing DLL procedures can be a recursive or even a circular process. Furthermore, DLLs can call any of the other service types listed in the table.

Table 39-2. Service Types Compared

Service Type	Availability to Caller	Loading Time	Link Mechanism	Shared Code	Reentrant Code
DLL Module	Subroutine Call	Program Load or Runtime	Name	Yes	Yes
Object Module	Subroutine Call	Program Load	Name	No	No
System-Common	Subroutine (Call Gate)	SysInit	System-Common Number	Yes	Yes
Request-Based	Task Switch	SysInit	Rq Code	Yes	No

Statically Linked Object Modules

An object module is linked statically to a program. Static linking incorporates the the code and data of the object module into the resultant run file before the run file is loaded into memory. This static linking is the conventional way modules in libraries get linked into programs on CTOS systems. It increases the run file size by the size of the object module(s). Furthermore, every program that wants to use an object module must link a copy of the module into its run file.

DLL Procedures

A DLL is not linked into a run file. Instead, it is contained in a separate file that is linked to the client when the client is loaded. Dynamic linking modifies the client references to the DLL to point to the DLL as it exists in memory. The DLL contains a list of *exports* (usually procedures) registered by name and runtime location. When the client (or *importer*) refers to the export name, the operating system quickly maps the name to the export address.

If changes are made to object module source, an application must be relinked with the updated version of the object module for the changes to take effect. A DLL can be updated without being relinked with applications.

DLLs can economize on memory usage. Not only are exports sharable but a DLL exists in memory only while its clients are active.

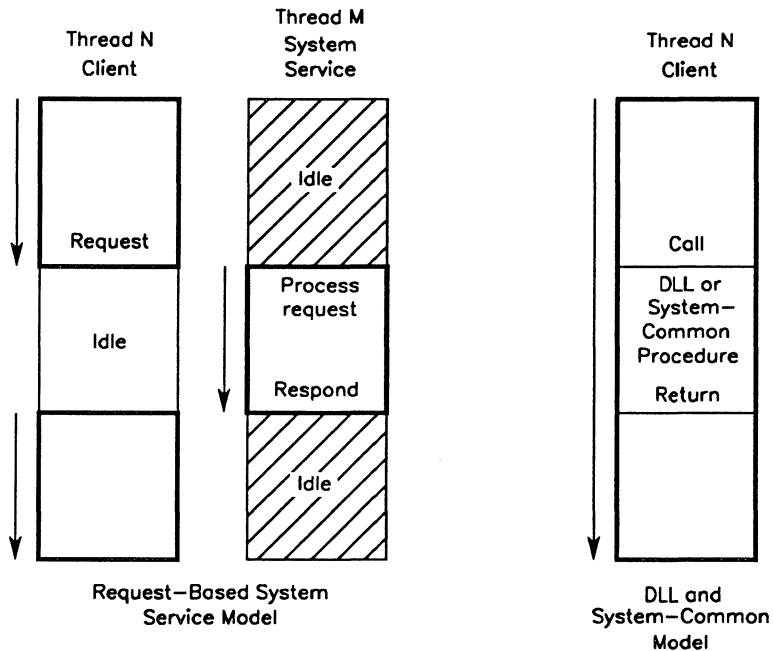
Comparison of DLLs to System-Common Services

DLLs are similar to system-common services in the following ways:

- Because neither of these services is request based, they cannot be distributed to remote clients in the cluster or over the network. You should design a request-based service if your intent is to distribute it.
- Both execute on the caller's stack. They run on the same thread as the client process rather than on separate threads. (See Figure 39-1.) It compares the execution model of a request-based service to a DLL and system-common service model.

- Both must be reentrant. The same caveats that apply to writing system-common services apply to writing DLLs. (See “General Guidelines for Writing DLLs.”)
- They act, essentially, as agents for the caller. Any resources they allocate are owned by the process (user number) that calls them.

Figure 39-1. Execution Models



557.39-1

DLLs differ from system-common services in two respects: the manner in which they are identified and whether they can be deinstalled.

DLLs are identified by name. The Linker lists DLL procedures the client will call in the client run file header by the name of the import procedure and the name of the DLL it is in. At program load, the operating system uses these names to determine if the DLL is in memory and to map client import references to the runtime addresses of the DLL exports.

DLL names do not collide with any other names in the system because the names are maintained in a private name space. The named addresses of procedures a DLL exports also reside in a private name space for that DLL. This arrangement allows identical procedure names to coexist in different libraries.

System-common procedures are identified by system-common number. When a system-common procedure is called, the operating system uses the number to determine where the procedure code is in the operating system. Because system-common numbers are managed by a central (human) source, there is a potential for system-common procedures to collide. Special care must be taken to ensure that each number is associated with a unique procedure.

DLLs are deinstalled automatically by the operating system when they have no clients. System-common procedures cannot be deinstalled: once installed, they become part of the operating system.

Request-Based System Services

DLLs share little in common with request-based system services other than both contain a single copy of code. Request-based services can share their code with other instances of the same request-based service. DLL code can be shared by several different clients.

How Dynamic Linking Works

The difference between dynamic and static linking becomes evident at the linking stage. Details on how the Linker resolves external references are described in the *CTOS Programming Utilities Reference Manual: Building Applications*. Enough of the linking process is presented here to convey the differences between the two linking concepts.

The Linker treats all external references in the same way, whether or not they are DLL procedures. When the Linker encounters a reference, it searches all the other object modules and library files included in the link for a matching public symbol. In the search, it may find either a conventional object file or an import library object module.

If the Linker finds the name of the procedure in question among the public symbols in a conventional object module, it performs a static link. To resolve the reference, the Linker copies the procedure code and data directly into the application run file and writes the procedure address to the address portion of the procedure call instruction in the program. (See “Object Module Libraries.”)

If, on the other hand, the Linker finds an import library object module, the module will not include the procedure code and data. Instead, it will contain the following information about the DLL procedure:

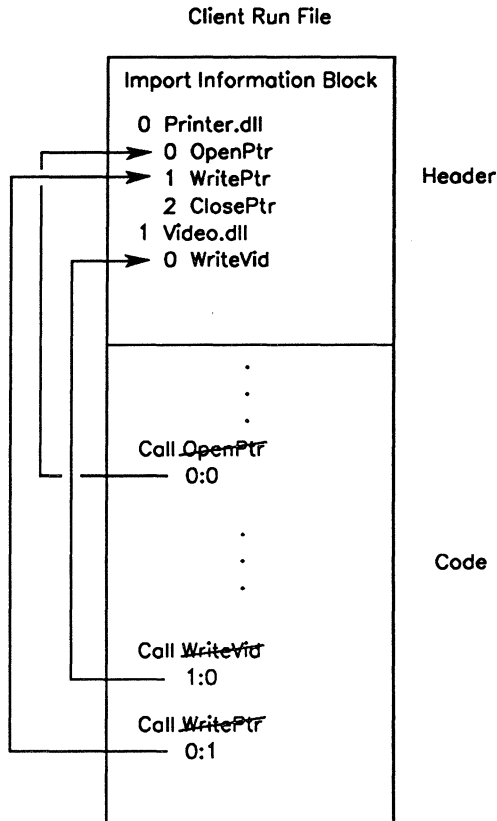
- The procedure name as specified in the client source file
- The name of the associated DLL

To resolve the reference, the Linker writes the above information about the DLL procedure to the Import Information Block in the client run file header. For each client reference to the procedure, the Linker provides the loader with information on how to locate the Import Information Block record. The loader uses this information to bind the importer to the exporter.

See Figure 39-2. It shows a client run file containing import procedures from two DLLs, Printer.dll and Video.dll. The Linker writes in the location of each import procedure (DLL number:Procedure number) at the call in the client code.

The Linker only creates an import entry for the first reference to a DLL procedure. The addresses of subsequent calls to the same procedure are connected to this entry in a linked list.

Figure 39-2. Imports



557.39-2

Impact on Loading

Because the code and data of statically linked procedures are already incorporated into the run file, the Linker can adjust addresses of procedure calls to point to the beginning of each procedure. The loader simply loads the run file and fixes up the local descriptor table (LDT) selectors to point to the runtime addresses in the loaded program. (See “Mapping Linktime to Runtime Addresses.”)

When a client calling dynamically linked procedures is loaded into memory, the loader examines the client Import Information Block. Any DLLs not already in memory must be registered by name and runtime location. Once the DLLs are registered, the loader can associate the runtime addresses of the exports with the client import references.

By examining the DLL segment types, the loader may determine it needs to load a separate copy of a segment for the current client. One copy of each DLL code segment is created for all clients to use. Data segments, however, may be shared among all clients, or each client can have its own copy. Depending on how the data segments are defined, the loader takes a different action. (For details, see “Sharing Data Among DLL Clients.”)

If the segment is defined as a *global segment*, it adjusts addresses to point to the single shared segment whose selector is in the GDT. If the segment is defined as an *instance segment*, it creates a local copy of the segment in the client’s memory. The selector for this segment is in the client LDT. (See “Defining DLL Segments” and “Assigning LDT Selectors,” for details on using the GDT and LDT for DLL segment types.)

Mapping Linktime to Runtime Addresses

The loader maps linktime addresses to runtime addresses. On virtual memory operating systems, this mapping process does not take place until the page containing the reference is actually used (code in it is called or data accessed).

Delaying address mapping is beneficial in two ways. First, the operating system performs the process only for references that are used. No performance time is wasted mapping unused references. Secondly, a page with mapped addresses is considered clean (unless it is modified in some other way). As such, it does not need to be written out to the disk. This reduces I/O time and saves disk space.

The loader maps the import references provided by the Linker to the runtime addresses of the corresponding DLL exports.

Link time addresses of DLL exports are first converted to runtime addresses in the “global selector space” (GDT selectors and certain LDT selectors). (See “Defining DLL Segments” and “Assigning LDT Selectors,” for details.) Because a DLL must be globally available, any other selectors it contains (for example, its internal references) also must be mapped to global selectors.

Module Definition File

The *module definition file* (commonly called a *.def* file) is a simple text file you create for each DLL. From this file the Module Definition Utility produces an object module containing supplementary information for the Linker and loader not provided by language compilers. Some of the module characteristics that can be defined in this file are listed below:

- Module name
- Segment attributes
- DLL export procedures
- Client import procedures
- DLL initialization procedures

The *module name* is the DLL name.

Segment attributes determine the types of segments that will be created for the program. A data segment, for example, can be defined as modifiable or read only. Segment attributes also determine whether a segment is copied to local memory (instance segment) or if one copy of the segment is shared among all client users (global segment).

For the module linked into the DLL, you define the names of the export procedures the DLL will make available to clients. Optionally, if you elect to have a client import the procedures by another name, you can define the names of the import procedures.

(For details on DLL initialization procedures, see “Executing Initialization Procedures.” Creation of module definition files is described in the *CTOS Programming Utilities Reference Manual: Building Applications*.)

Using the Module Definition Utility

The Module Definition Utility accepts a module definition file as input and produces an object file (*.obj* file) describing exports and/or a library of object modules each of which describes an import. A DLL is linked with an export object module, and an application, with the library of import object modules.

(For details, see the *CTOS Programming Utilities Reference Manual: Building Applications*.)

Using Resources

A DLL is not a separate process from the client. As such, it does not have resources of its own other than selectors and memory in which to run. Like a normal subroutine, however, a DLL procedure has access to all the client's resources, such as memory and file handles.

Calling DLL Procedures

To call DLL procedures, the client simply declares the DLL procedures and calls them as it would any ordinary library procedure. The DLL containing the appropriate export is bound to the client at the time the program is loaded.

The client must know the name of the DLL and the DLL procedure(s) it wants to call. As with any conventional library, the application procedure interface (API) information must be provided to the programmer who wants to use a DLL.

Sharing Data Among DLL Clients

One of the significant features of dynamic linking is that it supports segments that can be used by a single client or shared by all DLL clients.

A segment used by a single client is described in that client's LDT. This segment type is called an *instance segment*. It is also known as a nonshared segment. The operating system creates a separate copy of an instance segment each time a different client begins using a DLL.

Instance segments can be used for information that is significant to each client on an individual basis. If, for example, a client is updating a counter variable, and the client depended on the number of times it incremented the counter by itself, the counter data would only be meaningful if it were managed by that client. Another example might be a client writing to a window. If the client needed to know where it left off to pick up writing at that point sometime in the future, the data describing the cursor position would need to be instance data.

A *global segment* is described in the GDT. This segment type is also known as shared segment. The operating system loads a single copy of a global segment. A DLL procedure always accesses the same global segment, no matter which client process calls the procedure.

Global data is information available to all users. A status flag, for example, might be set by one client to open a file. The next client who needs to read from the same file can check the flag, find that the file is open already, and simply perform the read.

Note: *Access to global data typically requires using a semaphore. For details, see the section entitled “Semaphores.”*

Global and instance are segment attributes you can define in the module definition file. (See “Module Definition File.”)

General Guidelines for Writing DLLs

The guidelines for writing DLLs described in the paragraphs below address the following topics:

- Observing caveats
- Defining the DLL and client modules
- Executing initialization procedures
- Terminating clients

Observing Caveats

There are many similarities between DLLs and system-common services. (See “DLLs Compared to Other CTOS Services.”) Unlike request-based services which force serial execution, DLLs and system-common services are reentrant. Basically this means the code may be called by more than one process or thread. They are not a separate process from the caller. These facts make writing DLLs and system-common services more complex than writing other services. (For guidelines on writing system-common services, see “System-Common Service Writing Guidelines” in the section entitled “System-Common Services.”)

When you write DLLs, you should observe the following caveats:

1. You cannot assume the selector value of the stack segment (SS) is the same as the selector value of the *automatic data segment* (DS). Since DLLs and client programs are linked separately, they each have their own DS. When a DLL procedure is first called, it must reload the DS register with the selector for its own DS. Before returning to the client, it must restore the original DS value. The computation model is usually referred to as large model.

Note: *Procedures that use the medium model of computation (for example, the procedures in the standard operating system libraries) are not callable from large model DLLs without special preparation. For details on how to use medium model procedures in DLLs, see the appendix entitled “Using Medium Model Procedures in Dynamic Link Libraries” in the CTOS Programming Utilities Reference Manual: Building Applications. For details on computation models, see “Models of Computation” in Section 1, “The CTOS Programming Environment,” in the CTOS Programming Guide.*
2. Each time a client calls a DLL procedure, the DLL procedure code is shared but data may not be. Data sharing depends upon whether the module definition file defines the data segments as instance or global segments. To coordinate resources such as memory shared by multiple clients, you need to use a semaphore. A *semaphore* ensures use of the resource by one process at a time. (For details, see the section entitled “Semaphores.”)

3. A DLL should exercise care when allocating a client's finite resources. Using a client's memory, for example, can generate unpredictable results in the client program if memory is in short supply.
4. DLL exports must be far procedures. In C language, this means the procedure must be declared as 'FAR'.
5. Since DLLs run on the caller's stack, they should use the stack conservatively. Creating large arrays on the stack, for example, is not recommended.

Defining the DLL and Client Modules

Whether you are writing a DLL or a client that will use one, you must create a module definition file. The file defines the procedures a DLL has to export to a client and the DLL procedures a client will call. (See "Module Definition File.")

Executing Initialization Procedures

A DLL initialization procedure can be executed automatically when an application first references any part of the DLL.

There are two types of initialization procedures: global and instance. *Global initialization* procedures are executed once when the first client references the DLL. *Instance initialization* procedures receive control each time a new client first references the DLL.

You would specify global initialization, for example, if your DLL defined a set of print management procedures. Before the printing procedures are called, the printer device would need a one-time initialization. Instance initialization is appropriate if, for example, each client required a handle to access the printer device. To perform both types of initialization, you can use instance initialization, and then govern the type of initialization that actually takes place with a flag.

Initialization procedures are declared in the module definition file. (See "Module Definition File.") You can optionally specify the type of initialization (global or local). To control the order in which multiple initialization procedures are run, simply reference the DLL whose procedure you want run first, and so on.

Terminating Clients

When a client terminates either normally or because of an error condition, it is important that the DLL clean up any resources the client was using. In the initialization procedure, you also can provide a list of cleanup procedures to be run at termination. Using the `ExitListSet` operation, you can add procedures to the list or remove procedures. `ExitListQuery` allows you to examine the current list of procedures.

The cleanup procedures are dispatched in a priority order. (See the description of `ExitListSet` in the *CTOS Procedural Interface Reference Manual* for details on priority.) This prioritization allows DLLs that are clients of other DLLs to be terminated in the correct order. At the same time, it requires care when adding a cleanup procedure with `ExitListSet`. The procedure must not call any other lower priority DLL function because the called DLL will already have been terminated.

A cleanup procedure completes in one of the following ways:

- When it returns
- When it calls `Exit` or `ErrorExit`
- When it is aborted (for example, causes a general protection fault, stack fault, and so forth)

If a procedure exits with a nonzero status code or is aborted, an entry is made in the system log file.

Configuring Your System to use DLLs

CTOS offers configuration file options for handling DLLs. Although configuration details are described in the *CTOS System Administration Guide*, these options are also mentioned here because of their usefulness to DLL writers.

Handling Errors

One error handling feature allows you the option of continuing or discontinuing a DLL dynamic linking procedure at the time an error is detected. Some of the basic errors that can occur when a client requests a DLL are listed below:

- The loader does not find the DLL on disk.
- The loader finds the DLL but it is the wrong version.
- The DLL is the correct version but does not contain the requested procedure.

Any of these errors can cause the executing program to terminate. With the `:ContinueLoadOnError:` configuration option, however, you can elect to continue with the linking procedure or to stop the procedure immediately. Each of these options is discussed below.

Continuing the Linking Procedure

In some cases it is advantageous to continue the linking procedure. The program can execute up to the point of a faulty DLL or nonexistent DLL procedure. If the erroneous code is never called, execution can proceed without the program terminating. If the client cannot be bound to a DLL because of some error, all references to the DLL are replaced with 0. Any subsequent reference to address 0 causes a general protection fault. As a result, the program terminates with an error.

Discontinuing the Linking Procedure

Alternately, you can discontinue the linking procedure immediately if there are errors. In such a case, an error message identifies the faulty DLL. Temporarily, you can write your program code around such procedures to test good code paths.

Setting Up DLL Search Paths

With the `:LibrarySearchPath:` option, you can configure your system to look for a DLL in one or more paths. When an application requests a DLL procedure, the operating system examines the search path list in the order you specify the paths. The following search path entry, for example, directs the operating system to search for the DLL in the current path (`.`), and if the DLL is not found, to search `[Sys]<Sys>`.

`:LibrarySearchPath: (., (/Sys/<Sys>)`

Note: *Using multiple search paths may affect system performance.*

Defining DLL Segments

Unlike the OS/2 implementation of dynamic linking which uses the local descriptor table (LDT) exclusively to define DLL segments, dynamic linking under CTOS uses both the LDT and GDT.

This book assumes you are familiar with Intel architecture. Recall that the GDT describes segments accessible by all programs running in the system. Global or sharable segments are described by global selectors allocated from the GDT. Local segments and DLL instance segments are described by local selectors allocated from the LDT. An application's LDT describes the segments that application alone can access.

Assigning LDT Selectors

The discussion that follows explains the way CTOS uses the LDT. Although you do not need to know this information to write or use a DLL, it is presented here to complete the overview of CTOS DLLs.

There are actually two types of LDT selectors. CTOS assigns LDT selectors at the low address end of the LDT to a program's code and its own data. Selectors at the high address end are assigned to DLL instance data.

Figure 39-3. Assigning LDT Selectors

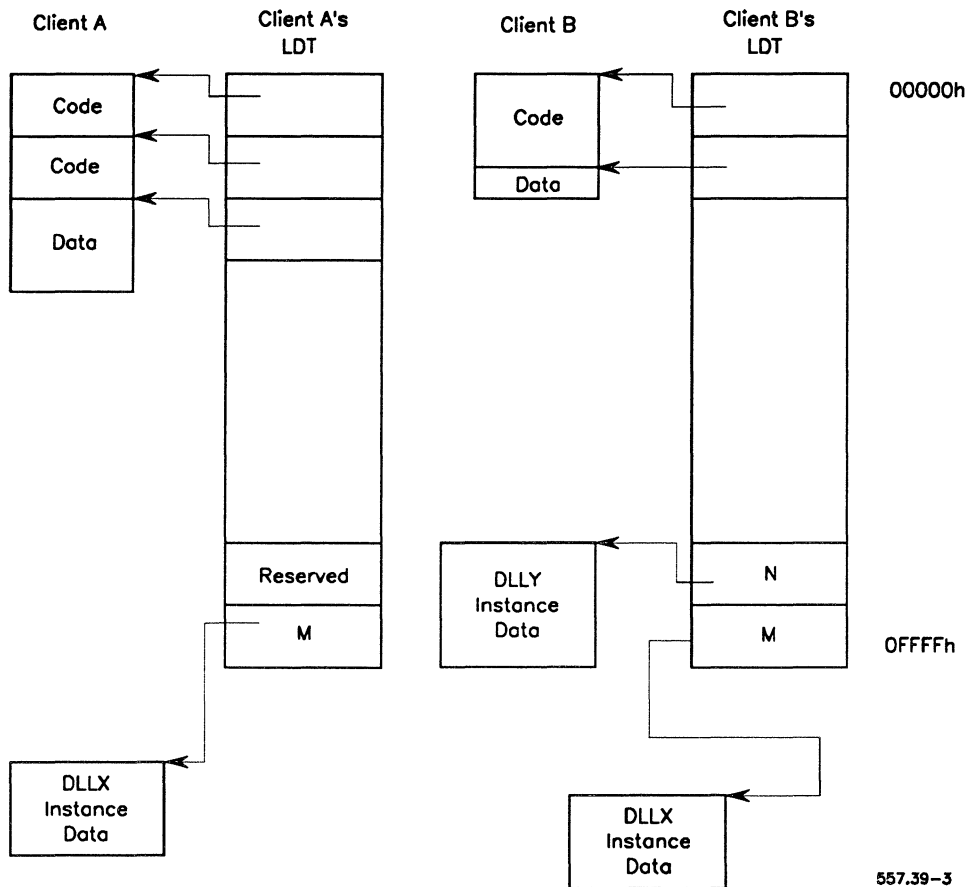


Figure 39-3 shows the LDT for Client A and Client B. Two LDTs are shown together in the figure for comparison purposes to illustrate the following points about the way CTOS assigns LDT selectors:

- The figure shows selectors being used at both ends of the LDTs. Client A and Client B reference DLL segments and have code and data segments of their own.

- The same selector in each LDT points to a copy of the same DLL instance data. Selector M points to the local copy of the instance data segment for dynamic link library DLL X.
- If a selector is assigned to a DLL instance data segment for one client, the corresponding selector is automatically reserved for the same segment in the LDTs of all subsequent programs. Selector N in Client B's LDT points to the DLL Y instance data segment. Selector N in Client A's LDT is reserved accordingly to point to a copy of the same data segment.

This method of assigning selectors allows DLL code to act upon unique data for each client while maintaining one way of addressing the data. Note the unused region in the middle of each LDT in Figure 39-3. While it is unused, it remains paged out so it does not consume any physical memory.

Summary of DLL Segments

Table 39-3 summarizes the way CTOS uses the LDT and GDT to describe DLL segments. DLL segments are distinguished by selector type, descriptor type, and linear address space. The descriptor type defines the access shown in the table.

Table 39-3. DLL Segments

Selector Type	Descriptor Type	Mapping Selector: Linear Addr	Access	Typical Use
sg	GDT	1:1	Global	Read only code and global data
sl	LDT (low)	1:1	Local	Local code and data
ssn	LDT (high)	1: several*	Clients attached to this DLL	Instance segments (usually data)

*1:several means one selector addresses unique memory for each client.

Special Use of Instance Segments

Upon loading a DLL containing initialization procedures, the DLL selectors are marked as not present. (The present bit is reset in the segment descriptor. For details, see the Intel documentation listed in “Related Documentation.”) When a client first references a DLL address, a segment not-present fault occurs. This signals the operating system to mark the segment as present and to run the initialization procedures.

The preceding paragraph more accurately describes how global initialization is performed than it does instance initialization. Global initialization procedures are run only once with the first client. (See “Executing Initialization Procedures.”)

Instance initialization runs initialization procedures for each new client that references the DLL. To run initialization on a per-client basis, all the DLL segments are defined as instance segments. This enables the operating system to mark the not present indicator separately in each client’s instance of the segment descriptor.

Runtime Dynamic Linking

Runtime dynamic linking requires the client make explicit system calls to load a DLL. (See “Requests to Perform Runtime Linking.”) It offers the advantage of allowing a client to dynamically select one of several versions of a DLL. For example, a different DLL might be written to manage the printing procedures for each type of printer available. Based on end user input, the client program can select the appropriate printer DLL to load.

Requests to Perform Runtime Linking

The following set of requests are provided to support runtime dynamic linking:

- LibLoad
- LibGetHandle
- LibGetInfo
- LibGetProcInfo
- LibFree

To perform runtime dynamic linking, a program calls the `LibLoad` operation, supplying the name of the DLL to be loaded. `LibLoad` loads the DLL, and returns a library handle that can be used to refer to the library in subsequent runtime linking requests. If the specified DLL references other DLLs, and one of the other DLLs cannot be found or fails to load, `LibLoad` returns the name of the faulting module.

To obtain the DLL name or the file specification of a DLL associated with a specified library handle, a program can call `LibGetInfo`. Given the appropriate DLL name, the `LibGetHandle` operation returns the library handle. Clients can use `LibGetHandle` to determine if a library is currently loaded. If not, the operation returns status code 1106 (“Bad library name”).

To obtain the memory address of a specified library procedure, a client calls the `LibGetProcInfo` operation with the appropriate library handle and procedure name.

When a client no longer needs to access a DLL, it calls the `LibFree` operation. `LibFree` causes the operating system to free the DLL from its association with the client.

DLLs: In Conclusion

Like most software libraries, DLLs undoubtedly will undergo future updates to add enhancements. Perhaps the biggest advantage DLLs have over conventional libraries is that, once a client program has used a DLL service, it automatically begins using the latest version of the service without having to be recompiled or relinked. As long as you maintain the same user interfaces (create a compatible DLL version), you are free to improve implementation details without client programs ever knowing improvements have taken place.

Dynamic Link Library Operations

The DLL operations described below are categorized by use. Operations are arranged alphabetically in each group. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Runtime Dynamic Linking

LibFree

Notifies the operating system that the client no longer requires access to the DLL specified by the library handle.

LibGetHandle

Returns the library handle of the DLL having the specified name. If the DLL is not loaded, *LibGetHandle* returns an error.

LibGetInfo

Returns the DLL name or DLL file specification for the DLL specified by the library handle.

LibGetProcInfo

Returns the memory address of the specified procedure within the specified DLL.

LibLoad

Loads a DLL (V8 runfile format) and returns a library handle. Because loading one library may cause other libraries to be loaded, *LibLoad* also returns a string identifying any modules that failed.

Initializing

ExitListSet

Adds a procedure to (or removes a procedure from) the list of procedures to be run when the client terminates.

ExitListQuery

Returns the number of procedures to be run when the client terminates and the addresses of the procedures.

Section 40

Interrupt Handlers

To most programmers, interrupts are invisible events, handled automatically by system software. This section will be of interest primarily to systems programmers, communications programmers, and others concerned with handling low-level devices or program instruction errors.

Interrupt Handling Terminology

The Intel microprocessors, upon which the operating system is based, support an interrupt handling mechanism that can be used for a variety of different purposes. For this reason, CTOS systems support a number of interrupt handling styles, some of which are only very distantly related.

To clarify differences in interrupt handling styles, we first define key interrupt handling terms used in this section. Most of these terms are not specific to operating system concepts but are instead used for the Intel family of microprocessors. For your convenience, the terms are briefly described in Table 40-1. Following the table, each is taken up in greater detail. (A few terms used in CTOS documentation vary from the Intel usage, as noted in the text below.)

Table 40-1. Interrupt Handling Terms

Term	Definition
External interrupt	An event triggered by a condition external to the processor; a device interrupt.
Internal interrupt	An immediate result of an instruction the processor tried to execute; a trap.
Interrupt	One of several types of control transfer initiated by the processor because of an event requiring immediate attention.
Interrupt Descriptor Table (IDT)	A table of 256 entries corresponding to different interrupt sources.
Interrupt handler	The code that receives control when an interrupt occurs; a device (external interrupt) handler or a trap handler.
Interrupt level	An integer in the range 0 to 255 identifying the interrupt type.
Interrupt Vector Table (IVT)	The real mode version of the IDT.

An *interrupt* is one of several types of control transfer initiated by the processor because of an event that requires immediate attention.

An *interrupt vector table* (IVT) is an array of program addresses maintained by the operating system. When an interrupt occurs (in real mode), the processor hardware consults this table to decide where to transfer control. The table has 256 entries, each of which can correspond to a different interrupt source. All real mode interrupts are directed to an interrupt handling routine by means of this table.

An *interrupt descriptor table* (IDT) is the protected mode equivalent of the IVT. For the purposes of this section, the two types of interrupt table are equivalent: each table is a 256 entry array that functions to direct interrupts to interrupt handling routines. The table used depends on whether the processor is in real mode or protected mode when the interrupt occurs. If the operating system does not support the use of both modes, only one or the other actually is present.

An *interrupt handler* is the code that receives control when an interrupt occurs. The entries in the IVT (or IDT) identify interrupt handlers.

An *interrupt level* is an integer in the range 0 to 255 that identifies the interrupt type (source of the interrupt). When an interrupt occurs, the hardware recognizes the interrupt type and the applicable interrupt level. The processor uses this value as an index into the IVT (or IDT).

Figure 40-1 shows the interrupt hierarchy. Each interrupt category includes one or more interrupt types.

The top-level categories are external interrupts and internal interrupts.

An *external interrupt* is an event triggered by a condition external to the processor. A peripheral device in need of service and a key pressed on the keyboard are examples of conditions that result in external interrupts. An external interrupt occurs asynchronously with the execution of the processor's instructions. It, therefore, can occur at an unpredictable time and usually is not related to the currently executing program.

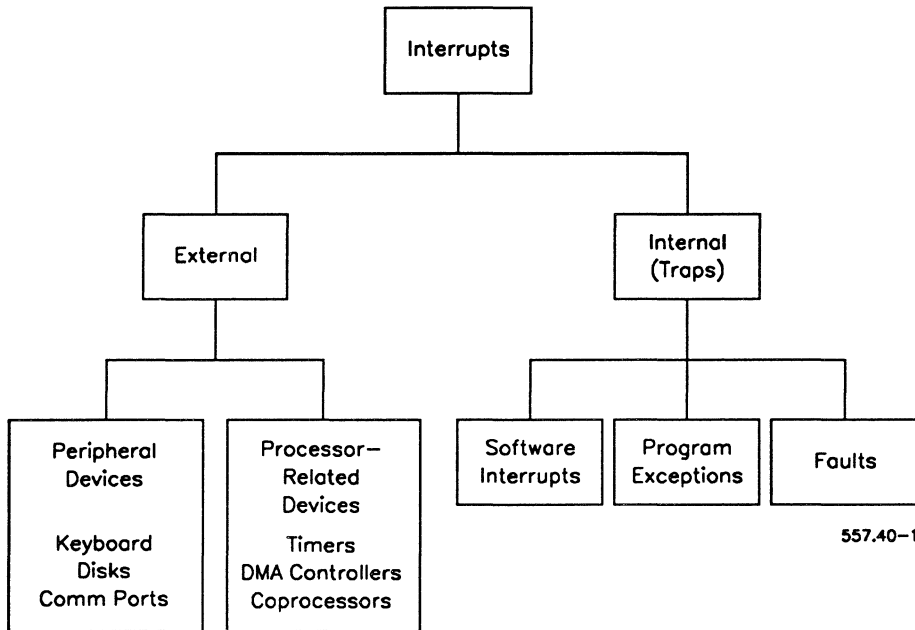
Device interrupt is synonymous with external interrupt. This is because an external interrupt results from an external device signal.

Note in Figure 40-1 that a programmable timer and a DMA controller are categorized as external devices (even if they are integrated into the processor chip). These devices are considered external to the processor because they operate asynchronously (in parallel to the processor's instruction stream). Floating-point coprocessors also are considered external devices for this same reason.

An *internal interrupt* is an immediate result of an instruction the processor tried to execute. Internal interrupts occur because instruction execution cannot, or should not, be allowed to proceed normally. An invalid opcode and an erroneous divide instruction are examples of conditions that result in internal interrupts.

Internal interrupts are unrelated to external events and have fewer conceptual implications. They can involve one process encountering an unexpected condition in a program, such as a divide by 0, or a deliberate process action, such as the explicit use of the INT instruction. In principle, an internal interrupt appears to be no different from a subroutine call.

Figure 40-1. Interrupt Hierarchy



557.40-1

In CTOS documentation, internal interrupts are also called *traps*. Traps are often handled by system services or application programs. Through system calls to the operating system, an application can establish its own trap handler. On protected mode workstation operating systems handlers can be established for any of the interrupt levels except level 14 (page fault exceptions) and 205 (CTOS request interface), even if a device handler is already established at that level.

Note: *Interrupt handling terms in CTOS documentation are sometimes referred to by different names in the Intel manuals. A CTOS internal interrupt, for example, is an Intel exception. An Intel trap is one of several CTOS internal interrupt types. For details, see the Intel manuals listed in “Related Documentation.”*

External Interrupt Handling Model

An external interrupt generally is used to alert the processor to service an external device in a timely manner. External interrupts are managed by a general model that provides device handling and control over interrupt occurrence.

Device Handling

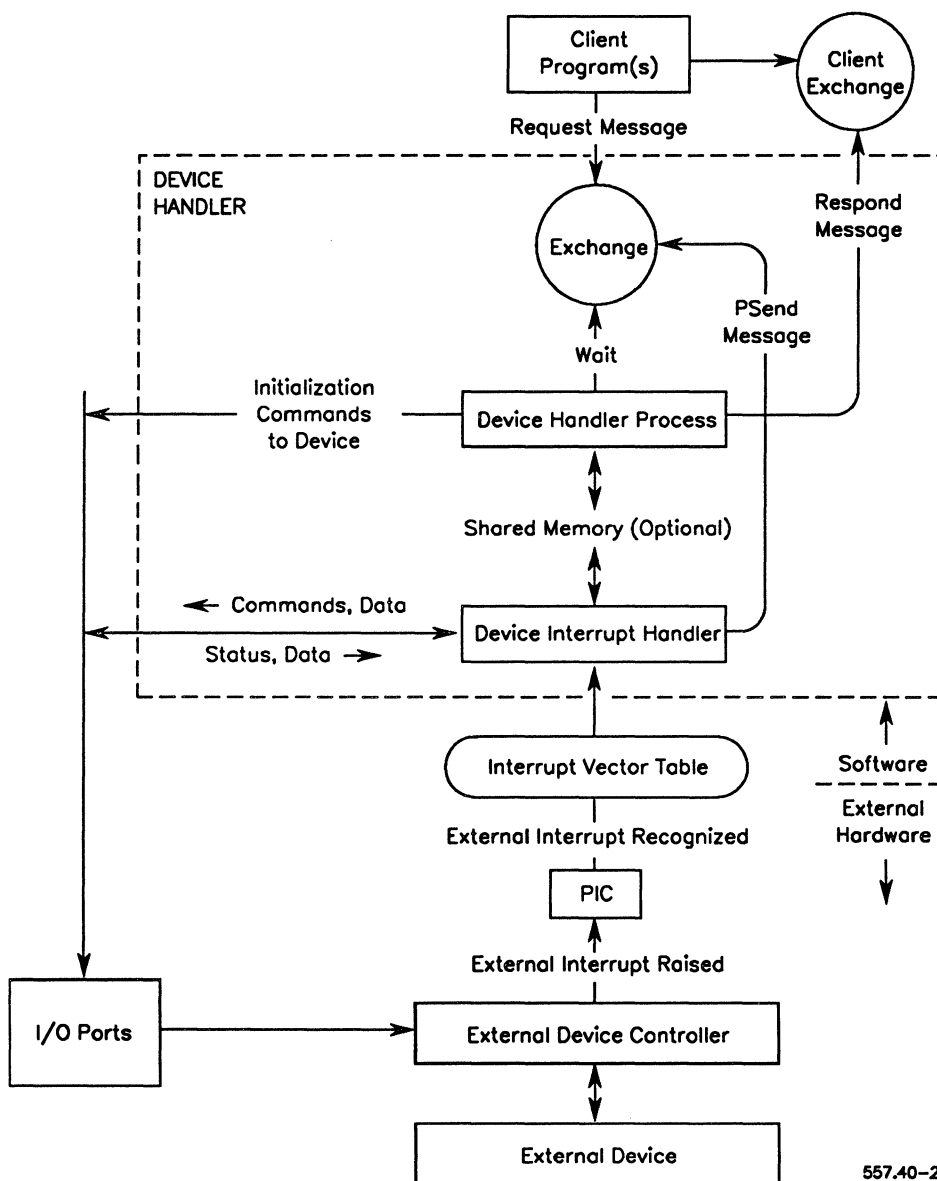
Device handling is accomplished by a device handler program. Device handlers perform the hardware I/O to and from an external device. Handlers for some devices are included in the operating system kernel; others are part of system services or application programs.

Device handlers usually consist of a *device handler process*, which manages the device and initiates I/O, and a *device interrupt handler*, which executes when operations complete or status conditions change at the device. Figure 40-2 shows a typical device handler.

Although they execute asynchronously (as if they were two processes), the device handler process and the interrupt handler are two closely related parts of the same program. Communication and synchronization are accomplished by using the PSend kernel primitive and, optionally, some shared memory, such as buffers and control information.

The device interrupt handler executes when the external interrupt occurs. If necessary, it may call PSend to start execution of the device handler process, which has been waiting at an exchange. PSend is effectively the only way the interrupt handler process and interrupt handler can synchronize. Only the device handler process (not the interrupt handler) may call the kernel primitive Wait to wait at an exchange, so it is impossible for the device handler process to use PSend to send a message to the interrupt handler. Synchronization, therefore, is unidirectional (from the interrupt handler to the process), although data communication can flow in either direction if shared memory is employed.

Figure 40-2. Device Handler



557.40-2

Device Handler Process

A typical device handler process spends most of its time idle. It waits at an exchange for either of two kinds of messages to reach it: commands from some program in the system that has work for the device, or messages (from its interrupt handler) that represent status or data from the device itself.

As such, the device handler process is both a clearing house for information related to the device and the agent responsible for determining what the device should do next. It is positioned between a client program using the device and the interrupt handler. (The interrupt handler, in turn, is positioned between the device handler process and the actual device.)

The device handler process does not run immediately when an interrupt occurs. It executes only if the interrupt handler sends it a message. Some interrupt handlers will send messages to their device handler processes each time an interrupt occurs; others do so only after a succession of interrupts have filled or emptied a data buffer. Devices that interrupt frequently enough can impede program performance to the extent that it would be prohibitively expensive to execute the device handler process after each interrupt. To maintain an acceptable performance level, the amount of work performed at each interrupt must be minimized. As an example, RS-232-C serial port devices cause frequent interrupts and, therefore, use an interrupt handler designed to optimize performance by avoiding the use of PSend on each interrupt. (For details, see “CRIHs and CMIHs.”)

Device Interrupt Handler

The device interrupt handler executes when an external interrupt occurs. It performs the following functions:

- As the primary responsibility, transfers data to or from the device or initializes DMA hardware that, in turn, performs such transfers
- Checks error and status conditions in the device after each interrupt
- In some cases, processes data

- In some cases (such as with RS-232-C serial port handlers), performs low-level protocol functions
- Decides when to start the device handler process executing (by calling PSend) to get further assistance

Because a device is prevented from causing additional interrupts while its interrupt handler is executing, the handler must service an interrupt expediently. Work that can be postponed (on input) or accomplished in advance (on output) should be performed by the device handler process, rather than the interrupt handler. Device handler processes typically can be interrupted, even by their own device interrupt handler, except during execution of critical code regions when the interrupt flag is turned off, disabling all external interrupts. As a result, device handler processes can take longer (than their interrupt handlers) to do their processing, without causing interrupts for the device to be lost. (See “Pending and Lost Interrupts.”) The interrupt handler often is programmed to buffer the I/O, effectively extending the time during which the device can transfer data before assistance from the device handler process is required.

Controlling When External Interrupts Occur

An external interrupt can occur after any instruction the processor executes. External interrupts, however, can be controlled by the interrupt flag and the Programmable Interrupt Controller (PIC) (described below).

The Interrupt Flag

Most external interrupts are maskable, which means that the processor can prevent them from occurring. This type of interrupt control is used, for example, to prevent interrupts while critical code regions are executing. Masking an interrupt is accomplished by clearing the interrupt flag in the flag word (disabling interrupts). Maskable interrupts can occur only when this flag is set (enabling interrupts).

When an external interrupt occurs, the processor hardware disables interrupts automatically. Certain interrupt handler styles allow the operating system to enable interrupts again before executing the interrupt handler; other styles keep interrupts disabled until the interrupt handler exits. (For details, see “Operating System Interrupt Handler Styles.”)

The Programmable Interrupt Controller

The programmable interrupt controller (PIC), a device closely associated with the CPU, extends interrupt enabling and disabling as implemented in the processor’s interrupt flag to multiple levels. It can be viewed as a part of the processor’s interrupt mechanism rather than a separate external device. (In some Intel microprocessors, the PIC is packaged as part of the same chip as the processor).

The PIC prioritizes interrupt signals from external interrupt generating devices and associates these sources with interrupt levels. Each device is wired to the PIC at a separate PIC input pin associated with a priority. Thus, hardware design fixes device priority. (Note that nonmaskable interrupt sources are the only type that is not wired to the PIC. For details, see “Nonmaskable Interrupts.”)

Devices with less patience are given a higher priority. Patience is the amount of time that can safely elapse before an interrupt is serviced by its interrupt handler. As an example, a hard disk drive has infinite patience. It can revolve forever while waiting to service an interrupt; the only penalty is increased rotational delay. On the other hand, a keyboard controller would require that the interrupt handler empty a one-character hardware buffer of a typed character before the operator pressed another key. Otherwise, an overrun would occur, because the buffer could not hold an additional character.

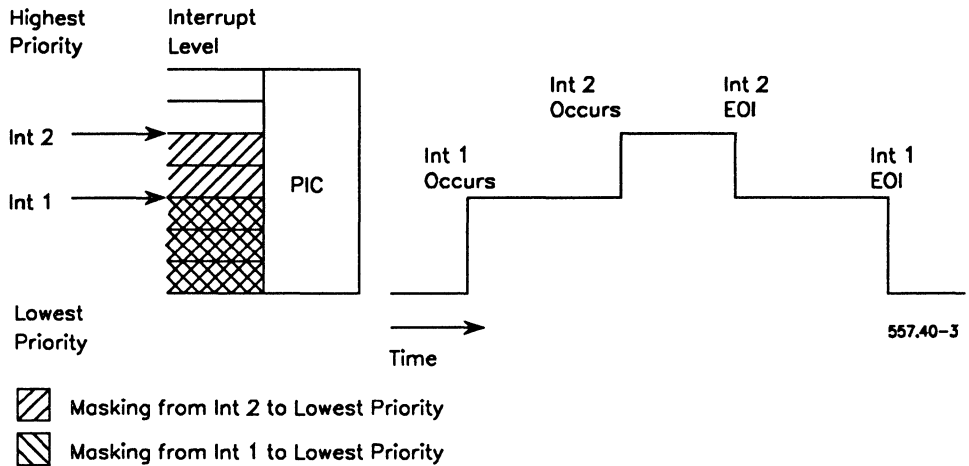
PIC management typically is an operating system function. Certain interrupt handler styles, however, require that the programmer issue PIC commands.

By issuing PIC commands, devices can be masked selectively. When an external interrupt occurs, the PIC automatically masks interrupts from the device causing the interrupt and from any other lower priority devices. Another interrupt from the same source cannot occur until the interrupt handler exits (even in cases where the operating system enables interrupts again before executing the interrupt handler). When the interrupt handler is ready to exit, the device is unmasked by sending an end-of-interrupt (EOI) command to the PIC. In some interrupt handler styles, the operating system sends the EOI command; in others, the command is sent by the user-written interrupt handler. (For details, see “Operating System Interrupt Handler Styles.”)

If the processor interrupt flag is set, the PIC’s selective masking can result in nesting of interrupt handlers for interrupts generated by devices of different priorities. Figure 40-3 shows an example of how this works. In the figure, the first interrupt (Int 1) results in the PIC masking that priority level and all lower priority interrupt sources. When the first interrupt is followed by a second, higher priority interrupt (Int 2), the PIC masks the higher priority level and all lower priority levels for the duration of the second interrupt handler’s execution. At EOI for the second interrupt, the second masking is removed, but the first one remains in place until EOI occurs for the first interrupt.

Device prioritization, therefore, ensures that devices may interrupt other device interrupt handlers with a lower priority, but not vice versa.

Figure 40-3. Interrupt Nesting



Pending and Lost Interrupts

If an event occurs that would cause an interrupt while that interrupt is masked or interrupts as a whole are disabled, that interrupt is not prevented entirely. It is merely deferred until it is enabled. Such an interrupt is called a *pending interrupt*.

Generally, one interrupt signal per device can be held pending at a time. If more than one interrupt signal occurs for the same device while the interrupt is disabled, only one of the interrupts ultimately occurs. Those that do not occur are *lost interrupts*. Such interrupts result in an overrun or underrun condition.

Most device controllers have special hardware to detect lost interrupts so that the device handler can report an I/O error. To prevent such errors, interrupts should only be masked for brief periods.

Nonmaskable Interrupts

A few external interrupt sources cause nonmaskable interrupts (NMIs). An NMI will occur regardless of the state of the interrupt flag or the PIC. It always causes interrupt level 2, which is dedicated to servicing NMIs on Intel microprocessors. All other interrupt levels are maskable when caused by an external source.

On the operating system processors, NMI sources include memory parity errors and bus ready timeouts. (The latter are caused by addressing a nonexistent peripheral device, or by a device for which initialization is programmed erroneously.) Even these NMI sources can be masked by writing appropriate commands to specific external hardware that controls them. Thus, there actually are no nonmaskable external interrupts.

Internal interrupts are never maskable.

Operating System Interrupt Handler Styles

The operating system supports two styles of interrupt handlers: raw and mediated. The two styles have different calling conventions and programming rules. They allow the programmer to decide between convenience or performance.

Note: *Virtual memory operating systems automatically lock the code and data pages of an interrupt handler into physical memory frames.*

A *raw interrupt handler* (RIH) offers performance over convenience in the following ways:

- The processor hardware transfers control directly to the user-written handler, which must be written in assembly language. (Coding in a high-level language defeats the purpose of writing the raw handler.)
- The handler must leave processor interrupts disabled. Because an RIH cannot be interrupted, nesting of interrupts cannot occur while it is executing.

Caution

In real mode, a raw interrupt handler executes on the stack of the currently running process. For this reason, a raw interrupt handler must carefully control its stack depth. A handler that uses in excess of 64 words of stack space can overwrite the memory of another process and cause system crashes with status code 22 ("Bus timeout"), 28 ("Invalid opcode"), or 91 ("Operating system checksum error"). In protected mode, each interrupted process (interrupt task) has its own stack. Stack overflow causes the system to crash with status code 92 ("Interrupt stack overflow").

RIHs are used for servicing high-speed, non-DMA devices.

A *mediated interrupt handler* (MIH) provides convenience over performance in the following ways:

- It can be written in a high-level language as well as assembly language.
- It permits automatic nesting of interrupt handlers by priority since processor interrupts are enabled during its execution. This means that a mediated interrupt is designed to be interrupted, if necessary, by higher priority device interrupt handlers.
- The operating system performs part of the handler's work by saving and restoring all registers and performing some or all of the EOI processing.

MIHs are recommended except for devices where interrupts occur so frequently that they would significantly impede program performance. Keyboard interrupts, for example, are serviced by mediated interrupt handlers.

Note: *Programs executing on GP boards of a shared resource processor must preserve the values in the remote slot and BR0 registers when doing inter-CPU transfers with mediated interrupts.*

Figure 40-4. Interrupt Handler Styles

	RS-232-C Serial Port Communications Interrupt Handler	Non RS-232-C Interrupt Handler
Raw Interrupt Handler	CRIH	RIH
Mediated Interrupt Handler	CMIH	MIH

557.40-4

There is also a difference in style between interrupt handlers for RS-232-C serial port communications devices and all other interrupt handlers. RS-232-C devices require special interrupt handling, because they cause frequent interrupts and, typically, several devices share an interrupt level. For these reasons and others, RS-232-C interrupt handlers have unique calling conventions and programming rules.

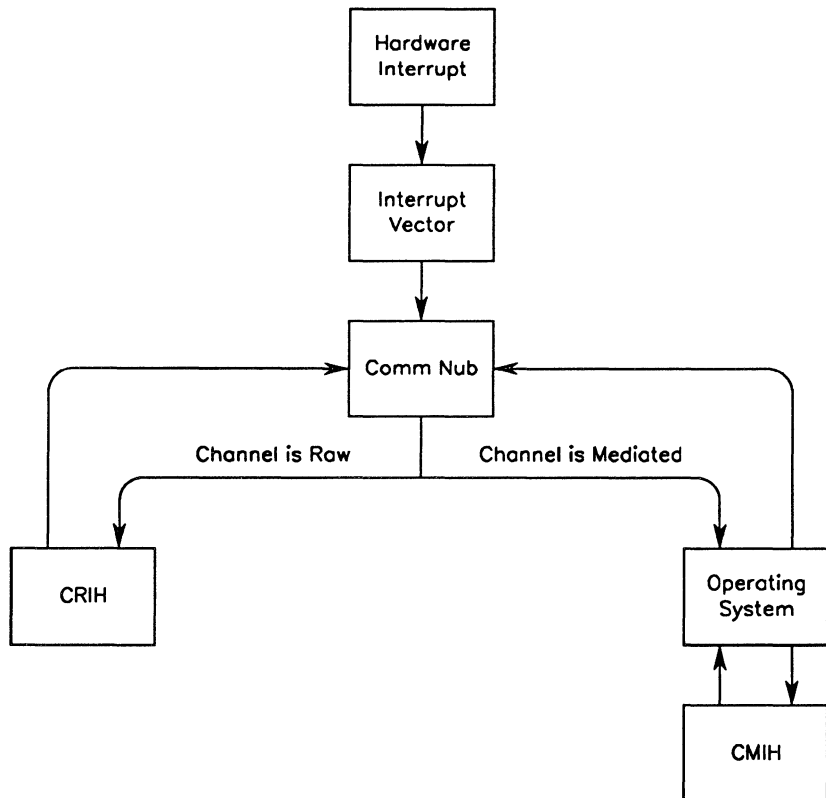
Differences in interrupt handling result in the four styles of interrupt handler shown in Figure 40-4.

Programmers concerned with RS-232-C devices should read the section on communications programming in the *CTOS Programming Guide* as well as the text below.

CRIHs and CMIHs

Figure 40-5 shows the program logic of a CRIH and a CMIH. The InitCommLine operation establishes the interrupt vector and the communications channel on that vector for the interrupt handler.

Figure 40-5. CRIHs and CMIHs



557.40-5

The *Comm Nub* shown in Figure 40-5 is a part of the operating system that dispatches CRIHs and CMIHs. A single hardware interrupt vector (PIC input pin) can support multiple communications channels belonging to different application programs. The *Comm Nub* directs the interrupt to its proper handler. It queries the serial controller's status to determine which channel is servicing the interrupt. Then, it determines whether the interrupt is a CRIH or a CMIH.

If the channel is serviced by

- A CRIH, the *Comm Nub* transfers control to the appropriate user-written CRIH. The user-written CRIH returns to the *Comm Nub* when it has completed processing the interrupt.
- A CMIH, the *Comm Nub* transfers control to the operating system, which in turn transfers control to the appropriate user-written CMIH. The user-written CMIH returns to the operating system, which then returns to the *Comm Nub*.

Guidelines for writing RS-232-C RIHs and RS-232-C MIHs are described below.

Guidelines for Writing a CRIH

To write efficient CRIHs, observe the following guidelines:

1. Interrupts must remain disabled for the duration of the interrupt.
2. In real mode, all processing is done on the stack of whatever process happened to be running at the instant the interrupt was taken, unless the CRIH requests that the *Comm Nub* call *PSend* to activate the device handler process. In such a case, after the CRIH returns, the *Comm Nub* switches to the operating system's interrupt stack before calling *PSend*. (The interrupted process is not resumed immediately if the awakened device handler process has a higher priority.)
3. A CRIH should use *PSend* to activate a device handler process only when necessary (not on every interrupt). This is because *PSend* overhead (process scheduling and context switching) usually exceeds the overhead of the rest of the *Comm Nub* and the CRIH itself. (See "Device Handler Process.")

The CRIH should communicate with its device handler process only as much as required. Usually the receive CRIH has a multicharacter buffer that it fills, and the transmit CRIH has a buffer that it empties before the device handler process is dispatched.

A typical error is to have the transmit CRIH activate the transmitting device handler process (which is waiting for buffer space) as soon as 1 byte of space is available. A better scheme is to have the transmit CRIH wait until the buffer is one-third to one-half empty. This avoids dispatching the transmitting process after each character sent, once the buffer is full.

4. Code a CRIH as tightly as possible. This code runs every time a character is sent or received: a few instructions can make a visible difference at a high baud rate or when multiple channels are in use simultaneously. Let the Comm Nub set up DS and BX so you can quickly locate the data structure needed to service the interrupt.

Figure 40-6 summarizes the guidelines for writing a CRIH.

Figure 40-6. User-Written CRIH Summary

-
1. All registers are saved for you.
 2. No parameters are provided on the stack when the CRIH is called.
 3. The Comm Nub sets DS to the selector of *pDsBx* and BX to the offset of *pDsBx*.
 4. Interrupts are disabled upon entry and must remain disabled for the duration of the CRIH.
 5. You do not do any of your own device controller or PIC EOI processing.
 6. You may not do a PSend on your own. The Comm Nub can call `MediateIntHandler` followed by a call to PSend for you. You may cause a PSend upon exit from the CRIH by leaving a non-zero value in the AX register. If AX = 0, no PSend occurs. If AX <> 0, AX is used as an exchange and DS:BX as the memory address of the message (pMsg).
 7. Exit using RET. All registers are restored for you automatically.
-

Guidelines for Writing a CMIH

The *CMIH* is very similar to an *MIH*. (For details, see “Guidelines for Writing an *MIH*.”) The following are a few ways in which the *CMIH* differs:

- The `InitCommLine` operation is used to allocate the interrupt vector (rather than `SetIntHandler`, which is used by *MIHs*).
- The entry in the IVT (or IDT) does not direct the interrupt to the entry point of the *CMIH*. Instead, the interrupt is directed to the `Comm Nub`.
- In real mode, the `Comm Nub` switches control to the operating system’s stack. (In protected mode, each interrupted process executes on its own stack.)
- The user-written *CMIH* can use both the `ReadCommLineStatus` operation and the `WriteCommLineStatus` operation, as well as the `PSend`, `SetTimerInt`, and the `ResetTimerInt` operations (used by *MIHs*).
- When the user-written *CMIH* is called, one parameter is supplied. The parameter *pDsBx* is user-defined but normally indicates which of the communications channels is being serviced by the *CMIH*.

Figure 40-7 summarizes the guidelines for writing a CMIH.

Figure 40-7. User-Written CMIH Summary

-
1. All registers are saved for you.
 2. One 4 byte parameter is supplied on the stack when the CMIH is called: the *pDsBx* provided to *InitCommLine*.
 3. DS is set to the selector of the *pDsBx* parameter of *InitCommLine* upon entry; SS does not match DS.
 4. Interrupts are enabled during the CMIH: you may disable them briefly, if necessary.
 5. You may use the *SetTimerInt* and *ResetTimerInt* operations.
 6. You may do *PSend(s)* on your own.
 7. You do not do any of your own device controller or PIC EOI processing. (This is performed by the Comm Nub.)
 8. Exit using *RET*. All registers are restored for you automatically. The values you leave in *AX* and other registers when you exit do not matter.
-

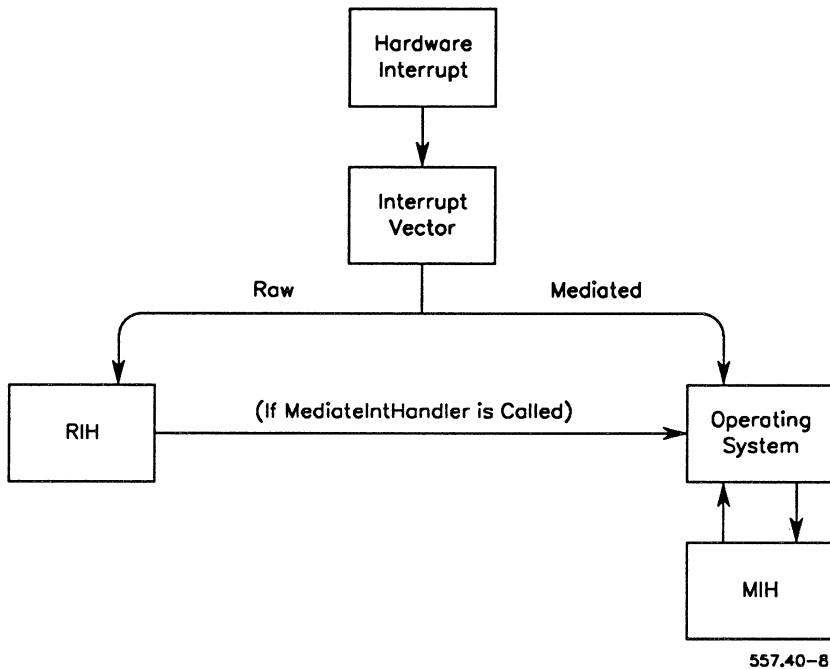
RIHs and MIHs

Figure 40-8 shows the program logic of an RIH and an MIH. The *SetDeviceHandler* and *SetIntHandler* operations are used to allocate the interrupt vector.

For ease in portation to systems with different hardware requirements, *SetDeviceHandler* is recommended over *SetIntHandler*. The caller specifies a device type that is constant across hardware rather than a variable interrupt type.

To detach an interrupt handler established by a call to *SetDeviceHandler*, an application can call the *ResetDeviceHandler* operation. By specifying a value of *FFFFh* for the device type parameter to this operation, the application can remove all interrupt handlers and device handlers established in previous calls to *SetDeviceHandler* and *SetIntHandler*.

Figure 40-8. RIHs and MIHs



Guidelines for Writing an RIH

The RIH must conform to the following rules. When an interrupt occurs, the RIH does the following:

1. In real mode it saves any registers that will be used.
2. It handles the device that generated the interrupt and processes as necessary.
3. It issues an EOI command to the device controller (if required by the device) and also an EOI command to the master PIC.

4. In real mode, it restores the saved registers.
5. It uses the IRET instruction to reenable processor interrupts while returning to the point of interrupt. In protected mode, the JMP instruction must follow IRET to transfer control to the beginning of the RIH.

The only operating system call an RIH can use is `MediateIntHandler`. It is used to convert the RIH to an MIH if the RIH determines that the device handler process needs notification for some reason. (For details, see “Guidelines for Writing an MIH.”) In this case, the RIH does not perform steps 4 and 5, above.

Figure 40-9 summarizes the guidelines for writing an RIH.

Figure 40-9. User-Written RIH Summary

-
1. In real mode, you must save all registers you use, because the only registers saved by the hardware are the flags, CS, and IP.
 2. No parameters are supplied when the RIH is called. (The RIH must be able to execute independently of parameters.)
 3. DS is set according to the *saData* parameter of `SetDeviceHandler` or by `SetIntHandler` upon entry; SS does not match DS.
 4. Interrupts are disabled upon entry and must remain disabled for the duration of the RIH.
 5. You must do all of your own device controller and PIC EOI processing.
 6. You must not make any system calls except `MediateIntHandler`. If you call `MediateIntHandler`, your RIH becomes a MIH (which can call `PSend`, `SetTimerInt`, or `ResetTimerInt`).
 7. Exit using IRET. (However, if you called `MediateIntHandler`, you must follow the MIH exit and termination procedures. For details, see “Guidelines for Writing an MIH.”) In real mode, you must restore all registers you use before you exit or call `MediateIntHandler`.
 8. In protected mode, IRET must be followed by a JMP to the beginning of the RIH.
-

Guidelines for Writing an MIH

Figure 40-10 summarizes the guidelines for writing an MIH.

Figure 40-10. User-Written MIH Summary

-
1. All registers are saved for you upon entry.
 2. No parameters are supplied upon entry.
 3. DS is set according to the *saData* parameter of *SetDeviceHandler* or by *SetIntHandler* upon entry; SS does not match DS.
 4. Interrupts are enabled during the MIH: you may disable them briefly, if necessary.
 5. If you used *SetIntHandler* to allocate the interrupt vector, you set the flag *fDeviceInt* in to TRUE; you do not do PIC EOI processing.*
 6. You may use the *SetTimerInt* or *ResetTimerInt* operations.
 7. You may do *PSend(s)* on your own.
 8. Exit using *RET*. The values you leave in *AX* and other registers when you exit do not matter. All registers are restored for you automatically.

*Both the PIC and the interrupting device require an EOI command for successful completion of interrupt processing or the system will hang.

Examples of External Interrupt Handlers

The text below describes external interrupt handlers for parallel ports and the X-bus.

Parallel Port Interrupt Handlers

The *SetLpIsr* operation establishes the printer interrupt handler [also called a printer interrupt service routine (PISR)] to process interrupts generated by parallel printer port interfaces.

PISRs can be linked to the system image and declared at system build. Alternatively, they can be linked with a dynamically installed system service or an application program and declared through the use of the *SetLpIsr* operation.

X-BUS Interrupt Handlers

Three levels of interrupt handlers are provided for X-Bus modules: XINT0, XINT1, and XINT4. XINT0 and XINT1 are nonshareable. XINT4 is shareable.

XINT0 and XINT1

XINT0 and XINT1 are for X-Bus modules that require a fast interrupt handler. The interrupt handler can be either raw or mediated by the operating system, as specified in the SetIntHandler operation.

Since the XINT0 and XINT1 interrupts are nonshareable, a system can be configured to have at most two modules that require these interrupts. X-Bus modules that require these fast interrupt levels must be able to use either, as instructed by software.

XINT4

XINT4 is set up for modules that can tolerate a slower latency.

This interrupt level is implemented by the Xbif system service as a chain of interrupt handlers that are invoked in a round-robin fashion whenever an XINT4 occurs.

Each interrupt handler is of type Boolean and returns FALSE or TRUE in the AL register of the microprocessor. TRUE is returned if the XINT4 was generated by the module that the interrupt handler services.

This protocol means that the interrupt handler for any X-Bus module must have a way to determine whether or not its module generated an interrupt.

The SetXBusMIsr operation is used to establish an XINT4 multiplexed interrupt handler [also called a multiplexed interrupt service routine (MISR)]. Additionally, SetXBusMIsr controls dedicated XINT1 interrupt handler allocation.

Pseudointerrupts

A pseudointerrupt shares an interrupt vector among several application programs.

Pseudointerrupts are implemented in software rather than in hardware. In this sense, they are not really interrupts. However, they are similar to interrupts in that they result in an interrupt handler being executed.

An interrupt handler activated by a pseudointerrupt executes in the same environment and has the same responsibilities and privileges as an interrupt handler activated directly by a hardware interrupt.

The programmable interval timer (PIT) uses a pseudointerrupt mechanism. The SetTimerInt operation can be used to establish a PIT pseudointerrupt handler to service timer pseudointerrupts. (For details, see “Programmable Interval Timer” in the section entitled “Timer Management.”) Pseudointerrupts, in this case, allow each of several software routines to function as though it has exclusive use of the high-resolution PIT.

In a system service, such as the Cluster Line Protocol Handler, the 3270 Terminal Emulator, or a user-written device handler for realtime data acquisition equipment, there is a need for concurrent high-resolution interval timing. Each of the three pseudointerrupt handlers performs the same logical (but not device-dependent) processing as if it were servicing an external interrupt from the PIT itself.

The Xbif system service uses the pseudointerrupt mechanism for interrupts generated by X-Bus modules. The XINT4 interrupt handler is implemented as a chain of interrupt handlers invoked in a round-robin fashion whenever an XINT4 interrupt occurs. (For details, see “X-Bus Interrupt Handlers.”) The QIC tape system service, CT-Net Ethernet media system service, and Telephone Service are examples of programs that use Xbif.

Internal Interrupts

An internal interrupt or *trap* is caused by instruction execution. Depending upon the type of internal interrupt, the instruction may or may not have completed successfully.

There are three major types of internal interrupt: software interrupts, program exceptions, and faults.

Software Interrupts

A *software interrupt* is caused by the program explicitly using the INT instruction. In some operating systems (notably MS-DOS), this is the standard way to transfer control to the operating system to request services. A software interrupt is simply a specialized type of subroutine call: typically, different interrupt levels correspond to different services, and arguments and results are passed in registers.

Application programs do not use software interrupts directly to request services of the operating system. However, some versions of the operating system make use of software interrupts internally (for example, interrupt level 205 is used by the request interface), and software interrupts are used when MS-DOS is run under the operating system.

Program Exceptions

A *program exception* is the processor's response to an invalid instruction that cannot be executed. Program exceptions include the following:

- Divide error
- Overflow (INT0 instruction)
- Bounds check
- Invalid opcode

A program exception usually indicates a program error.

Faults

A *fault* occurs in protected mode only when the processor detects a condition that calls for operating system intervention. There may be nothing wrong with the instruction being executed. In fact, it is sometimes possible for the fault handler to resume execution of the program after attending to the condition that caused the fault.

All the following are faults:

- General protection
- Segment not-present
- Stack exception
- Page fault

The segment not-present fault is an example of a processor-supported fault used by variable partition operating systems for segment swapping. An application program causes a segment not-present fault when it attempts to access a code segment currently not in memory. The not-present fault signals the operating system to read in the missing segment after which the program is resumed as if nothing had happened.

The page fault is supported on virtual memory systems, which use paging instead of segment swapping to manage virtual memory. (For details, see the section entitled “Demand Paging.”)

It is possible to restart a program after most types of faults, because the instruction that caused the fault was not executed. The saved CS:IP is the memory address of that instruction, not the one after it. After servicing the fault, the operating system returns to the program (using the IRET instruction), and the instruction is restarted. Provided the faulting condition has been removed, program execution proceeds normally.

Because faults are potentially restartable, fault handlers are transparent to a program; program exceptions, on the other hand, generally are fatal. Note that certain kinds of general protection faults do not follow this rule, because a general protection fault indicates a program error.

Faults usually fall into the category of internal interrupts that are handled by the operating system. Fault handlers are rarely user-written.

Trap Handlers

Although the operating system usually handles faults, program exceptions and software interrupts are often handled by system services or applications that install their own trap handlers.

A *trap handler* is an interrupt handler that is in effect only for the program installing it. Trap handlers apply to all processes of a user number. Other user numbers may install their own trap handlers, which perform different functions.

Some programming language runtime packages include trap handlers, for example, to handle divide error program exceptions. When a program written in such a language causes a divide error, the runtime package prints an error message or takes other action appropriate to the language.

On protected mode workstation operating systems, an application can install a trap handler for any of the interrupt levels (0 to 255) except for level 14 and 205, which are reserved for the operating system use.

Note: *For every interrupt level shared by a trap and a device handler, additional overhead is required to determine the interrupt source. The overhead may have an adverse effect on communications applications with little patience.*

The SetTrapHandler operation is used to establish a trap handler for the currently executing program in real mode or protected mode. In protected mode SetTrapHandler uses a 286 trap gate. A *trap gate* is an interrupt structure in the IDT that references the interrupt handling procedure. The 286 trap gate is compatible with a stack in 286 format (flags, CS, IP are pushed on the stack as words).

The Set386TrapHandler operation establishes a local handler using a 386 trap gate. The 80386 trap gate supports virtual 8086 mode (flags, CS, and IP are pushed on the stack as double words).

On protected mode operating systems, system default trap handlers exist for use by programs that do not establish their own local handlers. A system service may replace a system default handler using the `SetDefaultTrapHandler` operation, for a handler expecting 286 trap gate, and the `SetDefault386TrapHandler` operation, for a handler expecting a 386 trap gate.

The `ResetTrapHandler` operation replaces a local trap handler established by `SetTrapHandler` or `Set386TrapHandler` with the system default trap handler for a specified interrupt level.

`QueryTrapHandler` returns information about the current trap handler or device handler for the specified interrupt level.

Not all internal interrupts are handled by trap handlers. Fault handlers, like external interrupts, usually are installed by the operating system using `SetIntHandler`. This results in a handler that is in effect system-wide. As an example, the paging service on virtual memory operating systems calls `SetIntHandler` to set up the system page fault handler.

Packaging of Interrupt Handlers

Additional interrupt handlers can be linked either with an application program or with a system service. The system service can be linked with the system image at system build, or it can be dynamically installed.

The following operations are used to inform the operating system of the existence of an interrupt handler in an application program or in a dynamically installed system service:

- `InitCommLine`
- `SetIntHandler`
- `SetDefaultTrapHandler`
- `Set386TrapHandler`
- `SetTrapHandler`
- `SetLpIsr`
- `SetXBusMIsr`
- `SetTimerInt`

Application Program

Packaging an interrupt handler with an application program permits the interrupt handler to occupy memory only when the application program that needs it is in memory. Also, somewhat less effort is required to package the interrupt handler with an application program. Generally, an interrupt handler that is used only by one application program should be packaged with that program.

System Service

If an interrupt handler must be available continuously, even while one application program is being replaced with another, the interrupt handler must be packaged with a system service. An interrupt handler that supports a device attached to a server (on behalf of application programs executing in cluster workstations) must be packaged with a system service in the server (and also must use the formal Request/Respond model of interprocess communication). Packaging an interrupt handler with a system service reduces application program run file size, which would otherwise include the interrupt handler. Generally, an interrupt handler that is used by all or most application programs should be packaged with a system service.

Interrupt Handler Operations

The interrupt handler operations described below are presented alphabetically. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

ControlInterrupt

Enables, masks, or ends an interrupt for a specified device in a device independent manner.

DeviceInService

Returns the type of device being interrupted in a device-independent manner (protected mode only).

InitCommLine

Establishes an interrupt vector and the communications channel on that vector for a CRIH or CMIH.

MediateIntHandler

Converts an RIH to an MIH.

PSend

Is a kernel primitive that functions identically to the Send primitive but is used instead of Send in interrupt handlers.

QueryTrapHandler

Returns information about the current interrupt handler (device handler or trap handler) for the specified interrupt level.

ReadCommLineStatus

Can be used by a CRIH, CMIH, or an application process to query certain RS-232-C signals not defined in the serial communications controller.

ResetDeviceHandler

Sets up the default interrupt handler for the specified interrupt level. This permits an application to detach an interrupt handler established by a call to SetDeviceHandler. If the device type is specified as FFFFh, the operation removes all interrupt handlers and device handlers established in previous calls to SetDeviceHandler and SetIntHandler.

ResetTimerInt

Can be used by a CMIH or MIH to terminate the Timer Pseudointerrupt Block (TPIB) initiated by a SetTimerInt call.

ResetTrapHandler

Replaces a local trap handler established by a call to SetTrapHandler or Set386TrapHandler with the system default trap handler for the specified interrupt level.

ResetXBusMIsr

Purges an interrupt handler previously established using SetXBusMIsr.

Set386TrapHandler

Establishes a trap handler for 80386 microprocessor-based systems using a 386 trap gate. Set386TrapHandler is always raw and is part of the process context for all processes in a partition (as opposed to being system-wide).

SetDefault386TrapHandler

Replaces the system default trap handler with a user-written system trap handler to handle the specified software generated interrupt (trap). The handler expects a 386 trap gate.

SetDefaultTrapHandler

Replaces the system default trap handler with a user-written system trap handler to handle the specified software generated interrupt (trap). The handler expects a 286 trap gate.

SetDeviceHandler

Is like the SetIntHandler operation in that it establishes a raw or mediated interrupt handler for device-generated interrupts, but the caller specifies the device type (*tyDev*) instead of the interrupt type (*iInt*) as the first parameter and there is no *fDeviceInt* flag. SetDeviceHandler should be used instead of SetIntHandler for devices because it allows a program to be portable. Interrupt types vary across hardware but device types are constant.

SetIntHandler

Establishes an RIH or MIH. Unlike SetTrapHandler, SetIntHandler disables swapping of the caller and is always in effect system-wide.

SetLdtRDs

Sets the local descriptor table register (LDTR) and DS registers of the caller in protected mode. SetLdtRDs also updates the LDT field in the caller's task state segment so that the LDT selector is preserved across a task switch.

SetLpIsr

Establishes the printer interrupt service routine (PISR) to process interrupts generated by the parallel printer interface.

SetTimerInt

Can be used by a CMIH or MIH to establish a PIT pseudointerrupt handler.

SetTrapHandler

Establishes a trap handler in real mode or protected mode. In protected mode, SetTrapHandler uses a 286 or higher trap gate. It is part of the process context for all processes with the same user number (as opposed to being system-wide), and it does not disable swapping of the caller.

SetXBusMIsr

Establishes an XINT4 multiplexed interrupt handler. SetXBusMIsr also controls the allocation of dedicated XINT1 interrupt handlers.

WriteCommLineStatus

Can be used by a CRIH, CMIH, or an application process to raise or lower certain RS-232 signals not defined by the serial communications controller.

Section 41

X-Bus Management

What is an X-BUS?

An X-Bus is a high-speed bus over which system modules or cartridges and the workstation processor module can interact programmatically. The operating system supports two X-Bus types: the intermodule, general-purpose expansion bus (X-Bus) and the X-Bus+. On SuperGen Series 5000 workstations, the X-Bus+ is used for communication among X-Bus+ cartridges; the X-Bus is supported for some modules.

(See your hardware manuals for specific details on X-Bus and X-Bus+ configurations.)

Note: *The input device bus (I-Bus) is a different bus type connecting input devices to workstations. (For details on the programmatic interfaces to I-Bus devices, see “I-Bus Device Management” in the section entitled “Keyboard and I-Bus Management.”)*

Locating the Base Address

The boot ROM in the main processor polls each X-Bus module or X-Bus+ cartridge and builds a table of IDs describing the workstation hardware configuration. Each module or cartridge is associated with a unique identification number (ID). On the X-Bus, the ID is a 16 bit number that uniquely identifies the module type. X-Bus+ cartridge IDs are 32 bit values. (For details on module and cartridge IDs, see the description of the GetModuleId operation in the *CTOS Procedural Interface Reference Manual*. Also see Appendix G in that same manual.)

Additionally, the boot ROM assigns each module or cartridge blocks of base I/O addresses. An application uses the base I/O address to perform I/O to the module or cartridge.

Dynamically Obtaining the Base I/O Address

To communicate with the specified module or cartridge, an application must determine its relative position on the bus. To locate the module or cartridge, an application can use the `GetModuleId` operation.

`GetModuleId` is called repeatedly, incrementing the bus position by 1 each time, until either the specified ID is found or status code 35 ("No module in this position") is returned. Having located the module, the application can call the `GetModuleAddress` operation to obtain its base I/O address.

Alternately, if an application needs to obtain an X-Bus module (but not an X-Bus+ cartridge) I/O address, the application can call the `QueryModulePosition` operation. `QueryModulePosition` does the work for your application of calling `GetModuleId` until the specified module ID is found.

Say, for example, a system service is installed to interface with a specified controller on the X-Bus. The system service must first determine if the right type of controller is present. To locate the specified controller, the service would need to call `GetModuleId` or `QueryModulePosition`. Then it would need to call `GetModuleAddress` to obtain the controller base I/O address.

`GetModuleId` and `GetModuleAddress` also can be used to determine the existence of an I-Bus device and its base I/O address.

Computing the Module Base Address

Calling `GetModuleAddress` is the recommended way to obtain the base I/O address of a module or cartridge. (See "Dynamically Obtaining the Base I/O Address.") You must use `GetModuleAddress` if you want to be able to port your application across all workstation hardware.

A second method of obtaining the base I/O address of an X-Bus module is to compute it. This procedure, however, is not recommended because it is only compatible with workstations prior to SuperGen.

On these earlier workstations, the boot ROM assigns base I/O addresses to X-Bus modules in the following order: the first module to the right of the processor is assigned addresses 100h through 1FFh, the next module to the right is assigned addresses 200h through 2FFh, and so on.

Based on the boot ROM address assignments, an application can compute the base I/O address of a specified module from its X-Bus position using the formula

$$\text{base I/O address} = 100\text{h} * (\text{position} - 1)$$

X-BUS Module/Processor Memory Access

There are three memory usage classes of X-Bus modules: master, slave, and master/slave.

An *X-Bus memory master* is a device that can access the processor RAM, but the processor cannot access the module's memory address space.

An *X-Bus memory slave* is a device that cannot access the processor RAM, but the processor can access the module's memory address space.

An *X-Bus memory master/slave* is a device that can access the processor RAM and in which the processor can access the module's memory address space.

Accessing X-BUS Module Memory

Note: *The X-Bus interfaces described below are supported for accessing X-Bus memory only. There are no comparable X-Bus+ interfaces.*

In most cases, the application programmer will be concerned with accessing X-Bus module memory. As an example, if you are writing a program that will manipulate the pixels of a bit-map workstation video display, some of your program instructions will require manipulation of the Graphics Controller memory.

Using X-Bus Operations to Access Memory

To access X-Bus module memory, your program must call the `MapXBusWindowLarge` operation, specifying the module and the amount of module memory needed to be accessed in that module. `MapXBusWindowLarge` returns the memory address(es) of the required number of contiguous, 64K byte segments.

`MapXBusWindowLarge` must be called at least once before the program attempts to access the module's memory. It must be called again if the program accesses a different module. On virtual memory operating systems, the first selector returned defines a huge segment. With the selector, your application can access all the module memory using 32-bit addressing. (For details on huge segments and 32-bit addressing, see "Segments" in the section entitled "Memory Management.")

`MapXBusWindowLarge` is compatible in real mode and in protected mode. There are, however, a few differences that you need to be aware of. These are described in the following sections:

- "Specifying a Window Size"
- "Accessing Modules in Protected Mode"
- "Accessing Modules in Real Mode"

`MapXBusWindow` is an older operation that performs the same function of providing access to X-Bus module memory. It returns the address of only one 64K byte segment, however. Because `MapXBusWindow` can result in programs that are not compatible in protected mode, it is recommended that you use `MapXBusWindowLarge` for all new programs.

Specifying a Window Size

Each X-Bus module that contains memory accessible by the processor must have an X-Bus window entry in the system configuration file or in the system generation prefix files. (For details, see the *CTOS System Administration Guide*.) The window may be 480K, 224K, or 96K bytes. At system initialization, the operating system determines the X-Bus window size of each X-Bus module.

For real mode, the operating system reserves a region of addresses at the end of the 1 megabyte processor address space at system initialization. The size of this region is the maximum X-Bus window size of all X-Bus modules attached to the workstation.

Accessing Modules In Protected Mode

Calling MapXBusWindowLarge in protected mode allows your program to access the memory of an X-Bus module. (See “Accessing X-Bus Module Memory.”)

From the viewpoint of the programmer, protected mode implementation of MapXBusWindowLarge is totally transparent.

MapXBusWindowLarge returns selectors for the amount of memory that your program specifies based on the *sWindow* parameter. Because protected mode provides a 16 megabyte or greater address space, it can accommodate mapping of X-Bus module memory to addressable memory regions above the first megabyte without the use of the extended address register (EAR). (See “Accessing Modules in Real Mode.”)

Accessing Modules In Real Mode

In real mode, calling MapXBusWindowLarge to access X-Bus module memory requires that the processor set up an *extended address register* (EAR). The EAR is used to map a portion of the main processor’s address space into the X-Bus memory address space instead of its own (and therefore decreases the address space of the processor). The real mode processor generates a 20 bit address (1 megabyte address space). To this address, the EAR adds an extra 4 bits. The X-Bus module is programmed to respond to this 24 bit address. Each module responds to a different 1 megabyte base address range out of the total 16 megabyte range, depending on its position in the X-Bus.

Note: *SuperGen Series 2000 and Series 5000 workstations do not have an extended address register.*

(For details on the EAR, see the hardware manual for your processor module.)

From the viewpoint of the programmer, real mode implementation of MapXBusWindowLarge reduces the address space available to the processor by the size of the largest memory window in the system. If, for example, the module with the largest window has a 480K byte window, the maximum memory available is 512K bytes. Additional memory beyond 512K bytes is invisible to the processor, as memory addresses between 80000h and F8000h are mapped to the X-Bus.

X-Bus DMA

A DMA controller in the processor module controls the transfer of data over the X-Bus from a memory master or master/slave to the main processor's memory.

All X-Bus memory master or master/slave modules other than disk and graphics devices use channel 1, mode 3 DMA (verify mode) when accessing the main processor's memory.

This arrangement is required for operation with protected mode operating systems and other X-Bus modules. The operating system initializes channel 1 DMA in this mode on powerup. Channel 0 is used by communications and channel 3 by the hard disk.

Communication and Start-Up Protocols

An X-Bus module may communicate with a program on the processor module through its I/O space and/or by using memory either in the module's address space or in the processor's address space.

If the communication is through I/O space or through a structure in the module's memory address space, additional programming steps are necessary to set up the communication.

If the communication is through a memory structure in the processor address space, the module must be informed of the structure address, as such structures cannot be at fixed memory locations.

The communications structure location can be given to the X-Bus module either by using the module I/O space, or by using a protocol that uses the X-Bus Initialization Structure (XBIS).

XBIS

The XBIF Service provides a standard way that intelligent modules can use to establish communication with software running on the processor.

The XBIS, a 16 byte structure at memory location 400h, provides an area in the main processor's memory in which a program can communicate with a memory master module. In general, a program electing to communicate with a memory master does the following:

- Reserves the XBIS using the LockXbis operation
- Initializes the memory master module
- Frees the XBIS structure using the UnlockXbis operation

This general procedure is exemplified below using Voice/Data Services and the Voice Processor module.

The Voice Processor module uses a private data structure for communication. To establish communication between the Voice/Data Services and the Voice Processor module, Voice/Data Services performs the following function sequence:

1. It calls LockXbis.
2. It places the Voice Processor module number in the XBIS data structure using the memory address returned from LockXbis.
3. It places the physical memory address (obtained by using PaFromP) of the Voice Processor's private data structure in the XBIS data structure.
4. It writes a value to the Voice Processor module base I/O address space (using Get ModuleID to obtain the base I/O address).

This causes the Voice Processor module to read location 400h to do the following:

1. Obtain the address of the private data structure.
2. Write a status byte to location 400h.
3. Interrupt the CPU.

Voice/Data Services then frees the XBIS with UnlockXbis.

X-Bus Interrupts

Three interrupt levels are provided for X-Bus modules: XINT0, XINT1, and XINT4. (For details on interrupts, see the section entitled, “Interrupt Handlers.”)

X-Bus Management Operations

The X-Bus management operations below are presented alphabetically. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

GetModuleAddress

Returns the I/O base address of the specified X-Bus module or X-Bus+ cartridge.

GetModuleId

Provides access to the workstation module identification tables that are constructed by the boot ROM for the X-Bus and the I-Bus.

LockXbis

Reserves the XBIS structure at location 400h. LockXbis returns the memory address 400h.

MapPhysicalAddress

Assigns a single selector to one or more physical I/O addresses (80386 and higher processors).

MapXBusWindow

Is the same as MapXBusWindowLarge. For protected mode compatibility, MapXBusWindowLarge should be used in all new programs.

MapXBusWindowLarge

Returns memory addresses for accessing the memory within an X-Bus module.

Mode3DmaReload

Sets up and programs DMA to and from X-Bus memory master devices using X-Bus mode 3 DMA.

QueryModulePosition

Determines the bus position of a module. The X-Bus or I-Bus, type code, and module number (if there is more than one module of the same type) are specified and the position is returned. QueryModulePosition only works with X-Bus modules.

ResetXBusMIsr

Purges a previously established interrupt handler using SetXBusMIsr.

SetModuleId

Sets the identification number of the specified module/cartridge on the input device bus (I-Bus), the X-Bus, or the X-Bus +.

SetXBusMIsr

Establishes an XINT4 multiplexed interrupt handler.

SwapXBusEar

Returns the word value that was previously written to the EAR. Any program using this operation is responsible for restoring the previous value when finished accessing X-Bus memory. (SwapXBusEar is needed in real mode only.)

UnlockXbis

Frees the XBIS structure for use by other programs.

UnmapPhysicalAddress

Deallocates a selector previously allocated by MapPhysicalAddress.

Section 42

Bus Address Management

What is Bus Address Management?

The system bus is the interconnector for transferring data to and from CPU memory and peripheral devices. To access the memory of a CPU, an application must use a bus address.

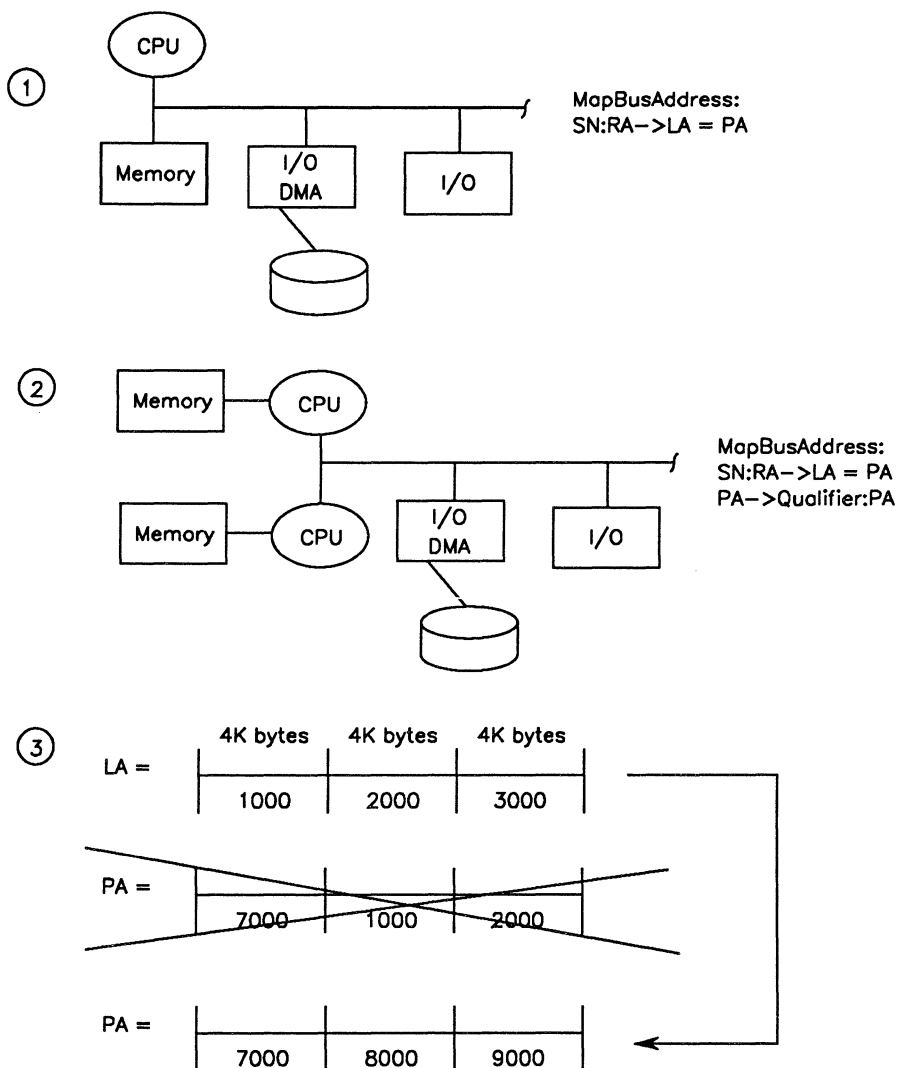
This section explains why bus addresses are needed and how they are formed using the bus address mapping operation `MapBusAddress`. (For details on accessing the memory in modules attached to a system X-Bus, see the section entitled “X-Bus Management.”)

To perform direct memory access (DMA) operations on virtual memory operating systems, there is a layer of preprocessing, even before bus addressability can be established. This preprocessing, accomplished by the DMA mapping operation `DmaMapBuffer`, also is explained in the discussion of why bus addresses are needed and is taken up in greater detail in “Using DMA Buffers.”

Why Bus Addresses?

A *bus address* is simply a value that enables an application to access a physical memory location on the CPU whose memory is being referenced. Given a specified selector and offset, the `MapBusAddress` operation generates a bus address that can uniquely identify the memory location regardless of the operating environment (type of bus, number of CPUs, or the operating system version).

Figure 42-1. Bus Address Types



See Figure 42-1. The bus address generated can be one of three types, as described below and shown in the figure:

1. On a variable partition operating system with a single CPU, MapBusAddress returns a linear address (LA), which is equivalent to the physical address (PA). With only one CPU, all physical memory addresses are unique. As such, the address returned by MapBusAddress is used as the bus address.
2. On a variable partition operating system with more than one CPU, the physical address alone is not enough information to identify a unique memory location. Each CPU is associated with its own local memory, and all local CPU address spaces start at address 0. In this case, the physical address requires additional information specifying the CPU. The operating system adds this *qualifying* information to the physical address to create the bus address.
3. To perform DMA operations on demand-paged virtual memory operating systems, some preprocessing must be performed prior to calling MapBusAddress. The DMA hardware requires a contiguous buffer in physical memory. Because physical addresses are not equivalent to linear addresses in a paged environment, this can't be guaranteed.

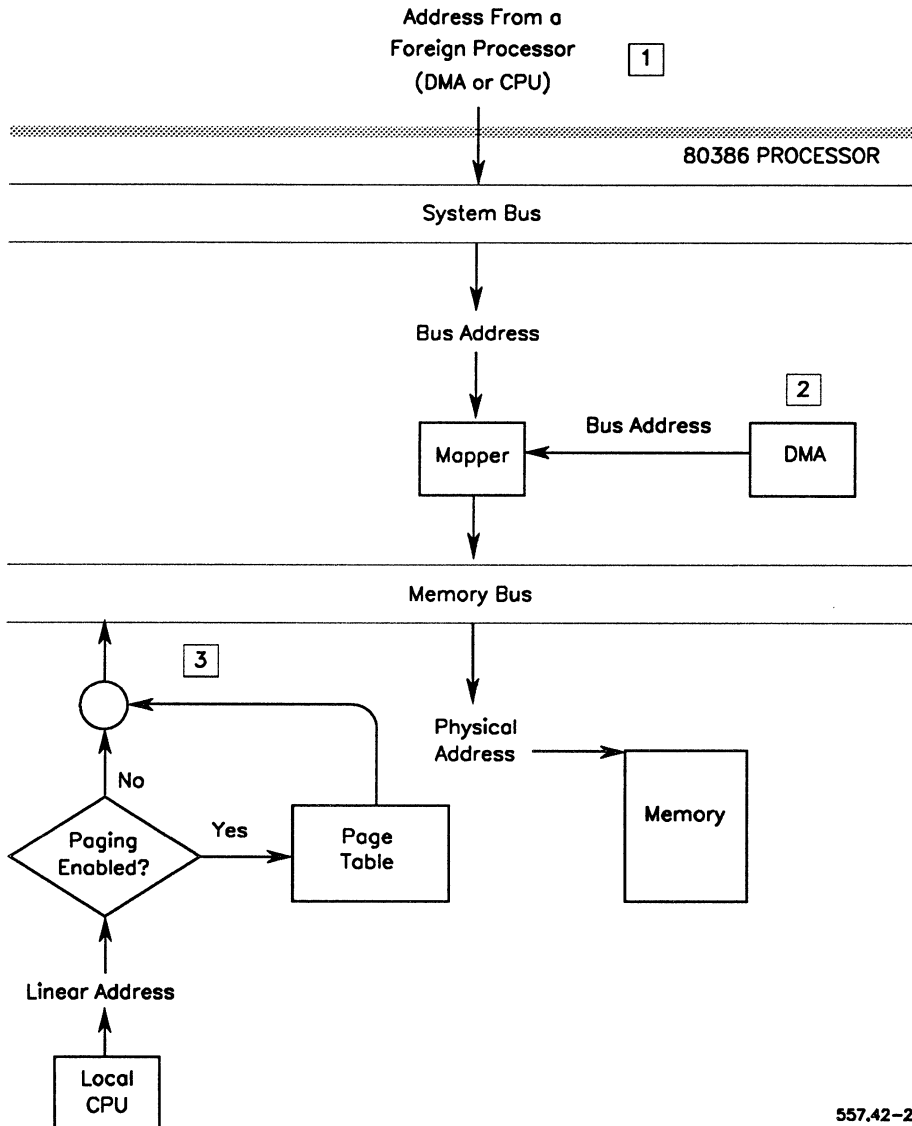
To ensure contiguity of the buffer, the application first must call a buffer mapping operation, such as DmaMapBuffer. As illustrated in Figure 42-1, the buffer can start at any physical address, but each 4K-byte page frame must be contiguous. (See "Using DMA Buffers" for details on DmaMapBuffer.)

With the DMA buffer in place, MapBusAddress can be called to obtain the bus address. Depending on the number of CPUs attached to the bus, the bus address is created with or without qualifying information, identifying the CPU.

Bus Addresses on an SRP

A shared resource processor (SRP) is one example of a system that uses a bus connecting several CPUs. Figure 42-2 shows the architecture of a single protected mode SRP processor board. The representation is highly simplified and serves only to illustrate how bus addresses are used.

Figure 42-2. Using Bus Addresses on an SRP Board



557.42-2

In the figure, an application executing on an external processor board references the memory on this processor board. The reference can be either a programmatic memory reference (denoted by CPU in the figure) or a DMA reference (denoted by DMA) to this processor's memory. An example of a programmatic reference is writing to the memory on this board by coding instructions to move bytes. (See "Using DMA Buffers" for details on DMA transfers.)

When the bus address arrives from the external processor, it is remapped to its corresponding physical address. This function is represented graphically by the arrows pointing from 1 to the box labeled "Mapper." The processor itself handles locally based, non-DMA references to memory on this board using its own transparent, internal mapping mechanism. This is illustrated at 3 in the figure.

Who Uses Bus Addresses?

Note: *Bus addresses cannot be used by real mode applications executing on protected mode operating systems.*

The operating system programs the DMA hardware to recognize bus addresses. If you are writing applications, you probably do not need to know the details of setting up or using bus addresses. To access the memory of another processor, your application simply provides the appropriate logical address, and bus address mapping is done transparently. If, however, you encounter an unusual address (like 0, which historically has been invalid) you should be aware that the address may be a valid bus address.

System programs such as the file system, the cacher, and inter-CPU communication (ICC) are the kinds of programs that typically need bus addresses to perform I/O between CPUs on behalf of other applications. At initialization, these programs call the MapBusAddress operation to obtain the bus addresses they need. Each SRP Kernel, for example, calls MapBusAddress at the time the ICC structures are created in processor memory.

Because the number of bus addresses is limited by the hardware, when a bus address is no longer needed, it can be recycled to the pool of available bus addresses for that processor by calling the `UnmapBusAddress` operation. (See the *CTOS Procedural Interface Reference Manual* for descriptions of `MapBusAddress` and `UnmapBusAddress`.)

Using DMA Buffers

An application can use the direct memory access (DMA) hardware to transfer data from one processor board to another or between a processor board and a device. To use the DMA hardware, an application must specify the address of a contiguous buffer in physical memory.

Because of paging on virtual memory operating systems, physical addresses are not necessarily contiguous with linear addresses. To ensure the contiguity of a DMA buffer in physical memory, an application can use either of two operations: `AllocCommDmaBuffer` or `DmaMapBuffer`.

`AllocCommDmaBuffer` is an object module procedure that is compatible across all versions of the operating system. It allocates the memory for a DMA buffer and maps the buffer to a contiguous area in physical memory. `AllocCommDmaBuffer` performs internal version checks on the operating system environment to ensure that the appropriate memory allocating and mapping calls are made.

Note: *`AllocCommDmaBuffer` may be used to allocate any DMA buffer. It is not restricted to use by communications applications.*

`DmaMapBuffer` is a system-common procedure available on protected mode operating systems. To obtain a contiguous DMA buffer with this operation, the caller first must make a separate call to allocate memory for the buffer. Then the caller provides the logical address of the buffer to the `DmaMapBuffer` operation. `DmaMapBuffer` maps and locks the buffer into a contiguous area of physical memory and returns its starting physical address. Once in memory, the buffer contents cannot be overwritten by the operating system. If `DmaMapBuffer` is unable to find a large enough area in physical memory to accommodate the contiguous buffer, it constructs one by swapping out page frames, as necessary.

Note: *For portability across all versions of the operating system, applications are encouraged to use `AllocCommDmaBuffer` or `DmaMapBuffer` over any other operation (such as `PaFromP`) to obtain physical memory addresses.*

Piecemealing the DMA Buffer

Occasionally, `DmaMapBuffer` cannot create a buffer large enough to accommodate all the data: the number of bytes actually mapped is less than the specified buffer size. This can happen, for example, if there are limits to the amount of data that can be transferred in a single DMA operation, or there just aren't enough contiguous page frames available for the buffer.

In this situation, the application must create two or more smaller buffers instead of the single larger one. To piecemeal a buffer, the application can call `DmaMapBuffer` a second time, incrementing segmented address and reducing the buffer size. The second call to `DmaMapBuffer` returns the physical address of another DMA buffer. Then the application can perform separate DMA operations using the physical buffer addresses returned.

Deallocating the DMA Buffer

When an application finishes a DMA operation, it can deallocate the buffer by calling `DmaUnmapBuffer`. `DmaUnmapBuffer` unlocks the area in memory occupied by the buffer and allows the operating system to reuse the memory.

Bus Address Management Operations

The bus address management and DMA buffer management operations are described below. Operations are arranged alphabetically in each group. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

Bus Addresses

MapBusAddress

Converts a logical address to a bus address and, on some systems, programs hardware necessary for the bus address to generate a reference to the appropriate physical address.

RemapBusAddress

Converts a logical address (in SA:RA form) to a bus address and, on some systems, reprograms the hardware necessary for the bus address to generate a reference to the appropriate physical address.

ReserveBusAddress

On systems that have address mapping hardware, *ReserveBusAddress* searches for and reserves hardware that may be subsequently programmed by a *RemapBusAddress* operation.

UnmapBusAddress

Invalidates a bus address created by an earlier call to *MapBusAddress*.

DMA Buffer Management

AllocCommDmaBuffer

Allocates the memory for a DMA buffer and maps the buffer to a contiguous area in physical memory. *AllocCommDmaBuffer* is an object module procedure that is compatible across all operating system versions.

DmaMapBuffer

Maps a linear DMA buffer to a contiguous, locked-in area in physical memory (protected mode operating systems only).

DmaUnmapBuffer

Informs the operating system that the specified linear addresses are no longer in use as a DMA buffer (protected mode operating systems only).

Section 43

Configuration Management

Configuration management instructs you in setting up the operating system.

System Administrative Actions

The following references describe system administrative actions primarily:

- The section entitled “Cluster Management,” describes cluster configurations and how the cluster works. It includes the cluster operations used in exercising administrative control over the cluster.
- For information on creating a bootable volume, see the *CTOS System Administration Guide* and your release documentation.

Programmer Actions

The following references involve programmer actions to reconfigure the operating system (rather than administrative actions):

- The section entitled “Native Language Support,” presents ways you can internationalize software to make it easily localizable in different native languages. The section also describes message files.
- The *CTOS System Administration Guide* and your release documentation describe generating a system (SysGen). SysGen consists of changing the default parameter values and/or removing functionality to build a customized operating system version.

Section 44

Cluster Management

What is Cluster Management?

Cluster management enables communication among cluster workstations and the server with which it is connected.

The *server* can be a server workstation or a shared resource processor (SRP).

Cluster Environment

One high-speed, RS-485 or RS-422 cluster communications channel is standard on each workstation. In cluster configurations connected to a server workstation, the server and all of the workstations connected to it use this channel for communications with each other. For large clusters with a shared resource processor server, multiple cluster communications channels are provided.

Each cluster channel is called a line and has a number associated with it. In a cluster configuration connected to a server workstation, there is one line, line 0. Shared resource processor clusters, however, have two lines per cluster processor board (GP+CI or CP) installed. The first board has line 1 and line 2, the second has line 3 and line 4, and so on. (There is no line 0 for a shared resource processor cluster.)

The cluster channel operates at various line speeds. For details, see your release documentation. Configuring cluster line speed is described in the *CTOS System Administration Guide*.

Status

The server keeps statistics about errors and normal operational parameters. The GetClusterStatus operation makes these statistics available to any program at any workstation.

The `GetClusterStatus` operation should be used instead of `GetWsUserName` to obtain the same as well as additional information about user statistics.

Polling

The server uses a technique called *polling* to check workstations that seek to use the RS-485 or RS-422 cluster line for cluster communications.

Polling starts every 50 milliseconds (one poll cycle) when a timer interrupt goes off, or whenever a response is ready to be returned to the workstation that initiated the request.

This method of polling

- Guarantees that a workstation will be polled at least once every 50 milliseconds when the cluster is not busy
- Polls active workstations more often than those not active

Roll Call

The server takes *roll call* by sending a message to each workstation that is currently online. If a workstation has a request to send to the server, it sends the message at this time; otherwise it informs the server that it has nothing to send.

The server notes which workstations had data to send during roll call.

Repoll

When the server gets to the end of the list of workstations it is polling, it checks to see if there is any time left in the poll cycle.

If there is time left, the server polls each workstation that was active again.

Repoll is repeated until a new poll cycle starts or no workstations were active in the last poll.

Polling is totally transparent to the programmer or workstation user. It appears, however, that the programmer/user is in control.

Request Routing Across the Cluster

Request routing depends upon how the request is defined and where the system service is installed. For further information, see the following sections:

- “Interprocess Communication,” for request routing by resource handle and file/device specification
- “Inter-CPU Communication,” for request routing between processor boards on a shared resource processor
- “System Services Management,” for defining requests to be used with user-written system services

If you follow the conventions for routing requests described in the sections above, not only will routing work correctly at your workstation, it will also work across the cluster or the network.

To have a request served locally, you must install the system service at your workstation, or status code 33 (“Service not available”) is returned.

The cluster operations described at the end of this section are used only in programs, such as the Cluster Status Utility (described in the *CTOS System Administration Guide*), to exercise administrative control over the cluster.

A request, such as Read or Write to a file server in a cluster, uses the interprocess communication (IPC) request/response model. (For details, see “IPC Summary” in the section entitled “Interprocess Communication.”) The same requests are used throughout the cluster. (Note that they differ only in the way routing is defined for them.) For this reason, there are no explicit operations in this section for communication over the cluster.

Cluster Management Operations

The cluster management operations are described below. Operations are arranged in a most to least frequent use order. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

GetClusterStatus

Returns usage statistics for each communications channel and the workstations attached to it.

QueryWsNum

Returns the number of the cluster workstation. QueryWsNum returns 0 if executed on a standalone workstation.

DisableCluster

Allows an application program on the server workstation to disable polling of the cluster workstations after a specified time period. DisableCluster is also used to resume polling of the cluster workstations.

FSrpUp

Returns TRUE if the cluster is operating.

GetClstrGenerationNumber

Returns information that may be used to determine the number of times the cluster has been regenerated after being disconnected from the server and whether or not remote handles are still valid.

Section 45

Native Language Support

What is Native Language Support?

Software is internationalized so it can be easily localized. Internationalization efforts result in applications that can be run in different native languages without modifying run file code. Localization efforts produce a single language definition. These nationalization capabilities of the operating system are known as native language support (NLS).

NLS, like keyboard management, is table driven. (See “Keyboard and I-Bus Management.”) In both areas, the programmer can localize software by customizing table contents to reflect a particular language.

Keyboard tables define alphabetic and numeric characters, strings, and diacritics. NLS tables with similar functionality are for backwards compatibility only with earlier operating systems. More importantly, NLS tables define other less obvious elements of a language definition, such as the format used to display the date and time, the criteria for comparing strings in the native language, and the native currency symbols.

Extended Native Language Support

Extended native language support (ENLS) facilitates nationalization by enabling character processing in native languages regardless of the character set size. Each of the 256 characters in the standard character set is uniquely defined using a single byte. Multibyte characters in extended character sets (exceeding 256 characters), however, require more than one byte for each character to have a unique definition. ENLS supports extended character sets. With ENLS, localization is made easy because applications do not need to be recompiled to process multibyte characters. They simply need to be relinked with the appropriate localized ENLS library. Currently ENLS supports 1, 2, or 4 byte characters.

Keyboard management also supports extended character sets. (See “Keyboard Management Features” in the section entitled “Keyboard and I-Bus Management.”)

Message Files

The message file facility is closely linked to NLS. Using message files stored outside the run file increases language independence in a program. An application can use any number of message files containing messages tailored to the native language.

NLS Terminology

Terms relating to NLS are described in Table 45-1.

Table 45-1. NLS Terms

Term	Definition
Nationalization	An umbrella term for internationalizing and localizing software.
Internationalizing	Making software localizable without having to alter source code.
Language definition	Unique language requirements, such as currency symbols and date/time formats.
Localizing	Making software reflect a language definition.
Message file	A file, separate from an application, containing messages the application displays.

How to Take Advantage of NLS

The NLS tables are in the Standard Software source file, *Nls.asm*. They control a number of different internationalizable aspects of software. Included among the tables, for example, are an uppercase to lowercase characters table, date and time formats tables, and a symbols table for numbers and currency.

The NLS tables allow you to nationalize operating systems in different ways. As shipped, *Nls.asm* defines U. S. standards. However, the file contains comments describing how the tables define standard single-byte character entries. The programmer can redefine table entries to suit the requirements of a native language simply by

- Editing *Nls.asm*
- Assembling *Nls.asm* to create *Nls.obj*
- Linking *Nls.obj* to create the system NLS file *[Sys]<Sys>Nls.sys*

(See the contents of *Nls.asm* for details.)

When the operating system is booted, it searches for *[Sys]<Sys>Nls.sys*. If present, the operating system loads the contents of this file into memory, making these tables available to application programs by means of programmatic calls.

Although you typically would want to use the NLS tables loaded at boot time, you can alternately create a different set of tables to link with your application. You can, in addition, elect to copy one or more tables into application memory. For details on these options, see “Using the NLS Tables.”

NLS Tables

Table 45-2 provides a brief overview of the NLS tables contained in *Nls.asm*.

Each of the NLS tables begins with a two-character (2 byte) *signature* to ensure table validity. The data for the table follows immediately thereafter. When the operating system loads the NLS tables, it verifies that the signatures of the tables it knows about are correct. Other tables can be added if desired.

Table 45-2. NLS Tables

Table	#	Signature	Size (Bytes)*	Use By
Keyboard Mapping	0	KE	varies	For backwards compatibility with earlier operating system versions
File System Case	1	FS	258	File system
Lowercase to Uppercase**	2	XT	258	EnlsCase, NlsCase
Video Byte Streams Text	3	VS	Varies (166 max.)	Video byte streams
Uppercase to Lowercase**	4	LW	258	EnlsCase, NlsCase, NlsULCmpB, ULCmpB
Keycap Legends	5	KC	Varies	GetNlsKeycapText
Date and Time Formats	6	DT	Varies	NlsStdFormatDateTime, NlsFormatDateTime
Number and Currency Formats	7	NC	9 to 11	NlsNumberAndCurrency
Date Name Translations	8	NT	Varies	GetNlsDateName, NlsParseTime
Collating Sequence	9	CT	Varies	NlsCollate

continued

*Size includes the 2 byte signature.

**This table is an n element array; n = size (bytes).

Table 45-2. NLS Tables (cont.)

Table	#	Signature	Size (Bytes)*	Use By
Character Class**	10	CC	258	EnlsClass, NlsClass
Yes or No Strings	11	YN	Varies	NlsYesOrNo, NlsYesNoOrBlank, NlsYesNoStrings, NlsYesNoStringSize
(Reserved)	12		0	
Special Characters	13	SC	Varies	NlsSpecialCharacters
Keyboard Chords	14	KC	32	For backwards compatibility with earlier operating system versions
(Reserved)	15	Varies		Reserved for keyboard mapping
Multibyte Escape Keys	16	MB	Varies	For backwards compatibility with earlier operating system versions
NLS Strings	17	CL	Varies	Used by operations such as OpenUserFile GetUserFileEntry SetUserFileEntry
Cluster-Share Keyboard	18	D1	258	ClusterShare
Cluster-Share Keyboard Extended Codes	19	D2	258	ClusterShare

continued

*Size includes the 2 byte signature.

**This table is an n element array; n = size (bytes).

Table 45-2. NLS Tables (cont.)

Table	#	Signature	Size (Bytes)*	Use By
Cluster-Share Video Translation	20	D3	258	ClusterShare
ClusterShare Key Post Values	21	D4	Varies	ClusterShare
OFIS Spreadsheet	22	none	512	OFIS Spreadsheet
Context Manager	23	UK	Varies	For backwards compatibility with earlier Context Manager versions
Gengo Date	24	GY	Varies	NlsStdFormatDateTime Formats, NlsFormatDateTime

* Size includes the 2 byte signature.

If an application obtains the table address using the table number, the address returned is the signature address.

Using the NLS and ENLS operations, your application can access the functionality of the NLS tables. In most cases, your program does not need to know the structure of any of the tables. If you write an application using the appropriate operations listed Table 45-2, the proper results are returned to your program for French, if there is a French *Nls.sys*, or German, if there is a German *Nls.sys*. If there is no *Nls.sys* file, all the operations return U.S. standards.

Note: *The NLS Keyboard Mapping table, Keyboard Chords table, and Multibyte Escape Keys table are for backwards compatibility only. Keyboard management does not use these tables for system-wide keyboard processing.*

The NLS tables contain the selections for all of the NLS options except for fonts and message text. (For details on using external message files, see “Message File Facility.”)

Detailed NLS Table Descriptions

Each of the NLS tables is described in detail in the paragraphs that follow.

Keyboard Mapping

Note: *The NLS keyboard mapping tables (NLS table number 0 and table number 15) are for backwards compatibility only. Keyboard management does not use these tables for system-wide keyboard processing.*

The *keyboard mapping* tables map keys pressed by the user to their character codes. With no diacritical key handling table number 0 is 386 bytes: the signature is 2 bytes, and the table is 384.

The *diacritical* key handling portion of this table defines characters with diacritical marks, such as the German ä. The first key of a diacritical key pair enables diacritical mode; the second key displays the diacritical result.

The length of the diacritical key handling portion can vary. It is determined by the following:

- the total count of diacritical keys (2 bytes)
- the diacritical key sequences [diacritic key pairs and their resultant values (3 bytes for each sequence)]

Nls.asm provides an example of how to edit keyboard mapping table number 0 to assign diacritical control to keys.

The total length of a keyboard mapping table is variable. In practice, however, the table will not ever be much larger than 400 bytes.

On workstations configured with *Nls.sys*, if both NLS table number 0 and table number 15 are present, a call to GetPStructure with structure code 270 will return table number 15 by default. You can override the default to obtain a pointer to table number 0 by using the :KeyboardProfile: system configuration option. (For details on workstation configuration file options, see “Configuring Workstation Operating Systems” in the *CTOS System Administration Guide*.)

File System Case

The *file system case* table is an optional table used by the file system for case-insensitive comparison and hashing. If the table is not present, lowercase Roman letters are mapped to uppercase Roman letters. No other characters are mapped.

It is recommended that you use only one such table definition on all systems. Problems can occur if you interchange file systems (for example, floppy disks) between systems with different language definitions.

Lowercase to Uppercase

The *lowercase to uppercase* table is used by EnlsCase and NlsCase to make case-insensitive comparisons and by other application programs to force a conversion of case. The OFIS Document Designer **Replace** command, for example, allows replacement control. If your application program requires collation, you should use the NlsCollate operation rather than using this table.

Video Byte Streams Text

The *video byte streams text* table is used by video byte streams. It allows translation of the following two prompts, which are displayed from within video byte streams:

Press NEXT PAGE or SCROLL UP to continue

Press NEXT PAGE to continue

Each of the above strings should be 80 or less bytes.

This table should need to be accessed only by video byte streams.

Uppercase To Lowercase

The *uppercase to lowercase* table is used by programs, such as EnlsCase, NlsCase, and NlsULCmpB, which must force a conversion of case.

This table can be used by programmers to translate characters.

Keycap Legends

The *keycap legends* table is used by `GetNlsKeycapText` to specify the text strings to be displayed by programs when referencing any of the keycaps commonly containing legends (for example, `GO` and `NEXT`).

(For a description of the keycap values, see `GetNlsKeycapText` in the *CTOS Procedural Interface Reference Manual*.)

Part or all of the keycap text may be translated. The maximum size allowed is 15 bytes of text (plus the 1 byte text string length). Any character codes can be used within the keycap names. It is recommended, however, that your program continue using the convention of displaying all uppercase letters for keycap text.

Date And Time Formats

The *date and time formats* table specifies format templates to control date and time construction. This table allows variations of date and time by country and by application program. (Also see “Gengo Date Formats.”)

The format strings serve as templates for the `NlsStdFormatDateTime` operation. This NLS operation substitutes the actual date and time for control letters embedded in the format strings. (See Appendix E, “NLS Templates,” in the *CTOS Procedural Interface Reference Manual*. It shows the control letters in the default NLS templates and provides examples of each template.) Control letters denote the various types of information used to construct the resultant date and time string.

Control letter order is significant. For example, whichever control letter appears first in the list is used to select AM, PM, Noon, or Midnight. Table entries should be in the case represented in the template. For example, the template below

`!Www! !Nnn! !2*A! !WWW! !NNN!`

would appear as

`Mon Jan PM MON JAN`

Number and Currency Formats

The *number and currency formats* table is used by the `NlsNumberAndCurrency` operation to control the formatting of numbers and currency fields.

Key elements in the table are presented in Table 45-3.

Date Name Translations

The *date name translations* table is used by `GetNlsDateName` and `NlsParseTime` to translate names of the months and days of the week.

More than one set of names can be defined. The format templates given to `NlsStdFormatDateTime` allow selection of which set of date names to use.

The maximum length of the date name string is 20 bytes.

(For a description of the index values used to reference the date names, see the `GetNlsDateName` operation in the *CTOS Procedural Interface Reference Manual*.) Names in this table should be lowercase. The format templates can be used to control selective conversion to uppercase.

Collating Sequence

The *collating sequence* table actually consists of four tables used by the `NlsCollate` operation.

The first table performs a simple substitution of character codes. This allows for reordering of the sort order including the mapping of uppercase and lowercase letters onto the same code values.

A second table defines 2-for-1 substitutions. Examples are the German ß, which collates as ss and ä, which collates as ae.

A third table defines 1-for-2 substitutions. Examples are the Spanish ch, which collates after c and Mc, which collates after M. In the second case, the name McGinty would collate after the name Morris.

Table 45-3. Number and Currency Formats Key Elements

Element	Description
decimalCh	A single ASCII character, either 2Ch (.) or 2Eh (.), used to indicate the decimal point in numbers. The default is 2Eh (.).
triadCh	A single ASCII character, either 2Ch (.), 2Eh (.), or 20h (space), used to indicate the separation of numbers into triads (that is, thousands, millions, and so on). The default is 2Ch (.). Note that use of space is not fully supported at this time. It may be ignored by some programs, or it may cause substitution of one of the other characters.
fFirstTriad	<p>A flag that controls the rules for placing the triad character in the thousands position. If TRUE, the triad separator in the thousands position always appears when the number contains four or more digits to the left of the decimal. If FALSE, the thousands triad separator is suppressed when no more than one additional digit appears to the left. This notation is commonly used in France. The default is TRUE.</p> <p>Note that some application programs never use triad characters, and others use them selectively or optionally. This flag merely controls the formatting when the program is using triad characters.</p>
listSepCh	<p>A single ASCII character, either 2Ch (.) or 3Bh (;), used to indicate the separation of numbers within a list. The default is 2Ch (.).</p> <p>Note that this specification is used only by application programs that would otherwise have a conflict with the use of 2Ch (.) as the decimal point character.</p>

continued

Table 45-3. Number and Currency Formats Key Elements (cont.)

Element	Description
iCurrencyPos	A value to control the position of the currency symbol. Zero (0) indicates the leading currency symbol; 1 indicates the trailing currency symbol. Other values are reserved for future expansion. Note that embedded currency symbols, such as 5\$33 to indicate \$5.33, are not currently supported.
sbCurrencySymbol	A string of up to 4 bytes containing the currency symbol. The first byte is the length of the string; the remaining 1 to 3 bytes contain the currency symbol.

The last (256 byte) table determines character priority. This table is used only when all prior tests have resulted in equality. This table is sometimes used for case differences, such as collating lowercase after uppercase only when otherwise equal; however, it is used more commonly for accent mark priorities. The vowel e, for example, is considered equal in all its forms except for priority, which alternates between uppercase and lowercase versions, first with no accents, then with acute, grave, circumflex, and umlaut.

Character Class

The *character class* table is used by `EnlsClass` and `NlsClass` to indicate the class of the character with the corresponding code.

Possible classes and their code values are as follows:

Class	Code
Numeric	0
Alphabetic	1
Special	2
Graphic	3
Blind	4

Graphic indicates that the character is used for line drawing or other special graphic purposes. *Blind* means that the character is not generally intended for display purposes.

Yes or No Strings

The *yes or no strings* table is actually two tables used by the NLS operations, `NlsYesOrNo`, `NlsYesNoOrBlank`, `NlsYesNoStrings`, and `NlsYesNoStringSize`. One table is a list of words meaning **Yes** in a particular language; the other table is a list of words meaning **No**.

Special Characters

The *special characters* table contains characters that are subject to special interpretation. An example of such a character is the literal insert, which is defined as 0A7h and is produced by `CODE'` on a U.S. keyboard.

The special characters table contains two regions: one for special character entries to be used internally and a second, for entries to be used by customers. A program can call the `NlsSpecialCharacters` operation to obtain the starting address of either of these table regions.

Keyboard Chords

Note: *The keyboard chords table is for backwards compatibility only. Keyboard management does not use this table for system-wide keyboard processing.*

The *keyboard chords* table defines the keyboard codes for the `CODE`, `SHIFT`, and `ACTION` key chords. The operating system recognizes the default function of these keys. This table can be edited to reflect differences in the physical positions of these keys such as may be found on a nationalized keyboard.

Multibyte Escape Keys

Note: *The multibyte escape keys table is for backwards compatibility only. Keyboard management does not use this table for system-wide keyboard processing.*

The *multibyte escape keys* table defines keyboard keys, which, when pressed, return multiple keystrokes to the keyboard client. Any key can be defined as a multibyte escape key, and different results may be defined depending on the shift and code key state. This table was used to generate nationalized strings on earlier versions of workstation operating systems.

Details on how to edit this table are contained in *NLS.asm*.

NLS Strings

The *NLS strings* table defines those strings used by operations in the standard operating library. Currently the following strings are defined:

.user
-old

Caution

Many applications use hard-coded strings rather than strings defined in the NLS strings table. In such cases, changing the .user suffix could cause the applications to malfunction. If you intend to redefine strings through the table, be sure to verify that your programs are compatible with the changes you make.

(This table is used by operations such as GetUserFileEntry, SetUserFileEntry, and OpenUserFile. For details on these operations, see “Parsing User Configuration Files” in the section entitled “Utility Operations.”)

ClusterShare Keyboard

The *ClusterShare keyboard* table maps single PC characters to the equivalent encoded character in the operating system's keyboard mapping table.

ClusterShare Keyboard Extended Codes

The *ClusterShare keyboard extended codes* table maps a PC key that has been pressed in combination with the control key to the equivalent encoded character in the operating system keyboard mapping table.

ClusterShare Video Translation

The *ClusterShare video translation* table maps an operating system font character to the equivalent visual character in the PC character set. (Currently ClusterShare Mail uses a similar table called *mpchCTchPC* located in the file *PcKbd.asm*.)

ClusterShare Keyboard Key Post Values

The *ClusterShare keyboard key post values* table performs non-unique character mapping between PC characters on two separate PC key posts that map to a single character in the standard character set.

The PC characters have different meanings, depending upon which key was pressed. For example, the hyphen (-) can mean the hyphen or **MARK** in the standard character set.

To distinguish between these keys, the key post value associated with the key is examined. For these non-unique character mappings, an assembler macro was created with the following parameters:

- The key post value
- The PC character
- The resulting character in the standard character set

When the particular key post value is encountered with the associated PC character, the program maps this keyboard input to the appropriate character in the standard character set.

OFIS Spreadsheet

The *OFIS spreadsheet* table is used to convert between the local operating system character set and the Lotus® International Character Set (LICS). LICS translation takes place when reading or writing Lotus compatible files. The NLS table is not necessarily required. (For further information, see the *CTOS OFIS Spreadsheet Reference Manual*.)

Context Manager

Note: *If the NLS Context Manager table is not needed, it should not be included in Nls.asm.*

The *Context Manager* table converts characters to their unencoded mode equivalents.

The Cut and Paste feature of earlier Context Manager versions used this table to ensure a correct Paste operation for applications reading the keyboard in unencoded mode. The current version of Context Manager, however, uses the keyboard Decoding table to perform this same function. (See “Translating” in the section entitled “Keyboard and I-Bus Management” for details on the keyboard Decoding supporting table.)

Each Context Manager table entry consists of a value representing a character in the standard character set. Following this value is the unencoded key sequence used to produce the character.

For example, to produce the uppercase character P (050h) in the standard character set, you press **SHIFT** and press **P**. Then you release **P** and release **SHIFT**. This sequence is shown below:

048h	(Press SHIFT)
070h	(Press P)
0F0h	(Release P)
0C8h	(Release SHIFT)

Gengo Date Formats

The *Gengo date formats* table specifies “emperor” year formatting. Emperor year formatting allows the year to be expressed as the number of years since the beginning of current emperors’ reign. Emperor years are used in Japan, for example.

The Gengo table provides format strings that serve as templates for the NlsStdFormatDateTime operation. (Also see “Date And Time Formats.”)

Using the NLS Tables

Through calls to the NLS operations your program can take advantage of the NLS tables. The first parameter to each NLS operation is *pNlsTableArea*. This is the memory address of the NLS tables to be used.

NLS Tables Loaded at System Initialization

Typically, an application uses the NLS tables loaded at system boot time. These tables are located in a *system-common NLS table area*. As such, they are available for use by all applications that want to use them.

The easiest way to access the system NLS tables is to pass NIL as the value of *pNlsTableArea*. NIL is interpreted as the base address of the system-common table area. When an application calls an NLS operation to perform a function such as comparing strings or translating names of the months and weekdays, the NLS operation, in turn, calls *GetPNlsTable* to obtain the offset of the specified table. *NlsCollate*, for example calls *GetPNlsTable* to obtain the address of the specified Collating Sequence table.

Alternate NLS Table Sets

Alternately your application can pass the memory address of a different set of NLS tables. The *OpenNlsFile* operation works with a set of NLS tables in an external file on disk. *OpenNlsFile* allows the caller to specify the name of the disk file. The operation returns the address to use as the value of *pNlsTableArea* in any of the NLS operations. This would be useful, for example, if you wanted an application to work correctly in two or more languages at the same time.

In certain cases, you may want to have an NLS table copied directly into your application memory. To do this, your program must call the *GetNlsTable* operation. *GetNlsTable* first searches for the specified table in the system-common NLS table area and loads the table into the application memory if it is found. Alternately, *GetNlsTable* opens the specified file and loads the contents into short-lived memory. It then copies the specified NLS table from the file to the application memory.

Internationalization

NLS facilitates writing applications that can easily be ported to localized operating systems provided native character sets do not exceed 256 characters. To be internationalized, an application should adhere to the following guidelines:

- It should use the appropriate NLS operation over the non-NLS equivalent. For example, applications should use `NlsULCmpB` or `NlsCollate` instead of `ULCmpB` to sort data. `NlsStdFormatDateTime` or `NlsFormatDateTime` should be used instead of `FormatDateTime` to format the date and time.
- It must use ENLS operations for processing multibyte characters. The NLS operations alone do not fully internationalize an application. If there is an ENLS equivalent to an NLS or non-NLS operation, the ENLS operation should be used. (For details, see “Extending Internationalization.”)
- It should use operations internationalized to access and use the NLS tables. Usually these operations have names with the prefix *Nls*. A few exceptions are the following internationalized utility operations:

`FormEdit`
`MenuEdit`
`OpenUserFile`
`GetUserFileEntry`
`SetUserFileEntry`

For details on these operations, see “Customizing the User Interface” and “Parsing Configuration Files” in the section entitled “Utility Operations.”

Extending Internationalization

To be fully internationalized, an application must not only use the NLS operations to take advantage of customized NLS tables but also the ENLS operations to process single-byte or multibyte characters.

ENLS operations are easy to identify. Like the NLS operation names, which begin with the prefix *Nls*, the ENLS operations begin with *Enls*.

The ENLS operations were designed to handle character sets containing more than 256 characters. The Japanese character set, for example, exceeds 6000 characters. Such character sets require more than one byte to define each character uniquely.

In many of the ENLS operations, what is passed is a pointer to a four-byte area in which the character (whatever its length) is written. The operations thus can support characters of varying length.

Applications using the ENLS operations can run without source changes on an operating system supporting character processing in a different language environment. The application simply needs to be relinked with the appropriate ENLS library.

If a native language defines multibyte characters, the ENLS library for that language typically reads one byte from the keyboard at a time until the entire character is read. How multibyte support is achieved in customized ENLS libraries is an implementation detail. The application writer only needs to know the support exists if ENLS operations are used.

The ENLS operations fall into three basic functional categories, namely

- Character processing
- String processing
- Line/form drawing

ENLS Character Processing

The ENLS character processing operations include

- EnlsCase
- EnlsClass
- EnlsGetChar
- EnlsGetCharWidth
- EnlsGetPrevChar
- EnlsMapCharToStdValue
- EnlsMapStdValueToChar

The character processing operations allow your application to perform functions such as editing a string of characters, determining the width of a character, and mapping characters to equivalents in other character sets.

To perform case conversions and character classifications, the `EnlsCase` and `EnlsClass` operations should be used instead of the equivalent NLS operations, `NlsCase` and `NlsClass`, respectively. The ENLS operations pass characters by address rather than by value. The NLS operations only pass single byte characters, limiting their use to processing single-byte characters.

The `EnlsGetChar` operation is of special significance. (See “Updating Applications.”)

`EnlsMapCharToStdValue` and `EnlsMapStdValueToChar` are designed to handle hard-coded dependencies on character codes. For example, an Executive command expects to read one-byte Yes/No responses from the command line. `EnlsMapCharToStdValue` maps a multibyte character to the equivalent one-byte character in the standard character set. `EnlsMapStdValueToChar` performs the opposite function by mapping a standard character to a non-standard one.

ENLS String Processing

The ENLS string processing operations include

- EnlsAppendChar
- EnlsDeleteChar
- EnlsFieldEdit
- EnlsFieldEditByChar
- EnlsFindC
- EnlsFindRC

The string processing operations allow your application to perform functions such as inserting or deleting characters from a string, locating the position of a character in a string, and appending a character to a string.

To edit a string of text, EnlsFieldEdit should be used instead of FieldEdit. EnlsFieldEditByChar is a variation of EnlsFieldEdit that provides greater control by allowing the caller to provide keystrokes as parameter data. EnlsFieldEdit reads keyboard input only.

EnlsFindC searches a string for a single-byte or multibyte character. EnlsFindRC searches a string in reverse order. Any time an application uses a high-level language library procedure to search for a character byte value, that procedure should be replaced by EnlsFindC or EnlsFindRC. As an example, EnlsFindC and EnlsFindRC should replace the PL/M procedures FindB and FindRB, respectively.

ENLS Line/Form Drawing

The ENLS line/form drawing operations include

- EnlsCbToCCols
- EnlsDrawBox
- EnlsDrawFormChars
- EnlsDrawLine
- EnlsQueryBoxSize

The line/form drawing operations should be used for drawing lines and boxes to the video or printer device, determining screen coordinates, and querying the memory requirements of a box in the video display. Like all the ENLS operations, these operations take character size into account, for example, to determine the cursor location or the size of the image drawn.

Usually there is a one-to-one correspondence between the in-memory character size (in bytes) and the display character width. However, there are exceptions. In the Japanese character set, for example, each form drawing character requires two bytes in memory but the display character occupies one column. To control alignment when drawing characters to the display device, an application can call the `EnlsCbToCCols` operation. `EnlsCbToCCols` returns the number of display columns required to display the specified character string.

Updating Applications

The `EnlsGetChar` operation is of special significance. Not only can it be used to read character input from a user-specified memory area, but also it can be used to update existing applications to read multibyte characters.

`ReadKbd` and `ReadKbdDirect` only read keyboard input one byte at a time. Historically, localizing an application to read extended character sets meant internally modifying the application run file to call these operations more than once to read a multibyte character.

By replacing these keyboard operations in an existing application with the `EnlsGetChar` operation and specifying the appropriate input stream type, `EnlsGetChar` calls the appropriate keyboard read operation once to read a single-byte character or as many times as necessary to read a multibyte character.

Localization

Localization allows operating systems as well as programs that run on them to reflect a particular language definition. Factors determining how a character is read can include

- Whether NLS tables are customized
- The version of the ENLS library linked with the application
- The character set present
- Whether the keyboard driver and VAM support the language definition
- The contents of the message files used by the application

(For details on message files, see “Message File Facility.”)

Customizing the NLS Tables

To localize an operating system such that it displays a particular language definition, you modify the source file, *Nls.asm*. To do this, you must change the applicable NLS table(s) contained in the file to meet the language requirements before assembling and linking to create *[Sys]<Sys>Nls.sys*. Thereafter, the system NLS tables reflect the modifications you made. (See “How to Take Advantage of NLS.”)

You can also localize selected NLS tables in *Nls.asm*. If, for example, the only requirement is to change the currency symbol from the U.S. dollar sign to the English pound sign, you can include only the Number and Currency Formats table in the NLS table area. This eliminates unnecessary tables, saving operating system memory.

You can, in addition, link copies of nationalized NLS tables with your application program. (For details, see “Alternate NLS Table Sets.”)

Linking With the Appropriate ENLS Library

An application must be linked with the appropriate ENLS library for the native language used. The standard version of the ENLS library contains all the modules needed to run programs processing single byte characters. To process Japanese characters requires linking with a Japanese version of the ENLS library.

Other System Customization Requirements

The keyboard driver and version of VAM installed can be standard or customized to meet local language requirements. Although these subjects are not described here in detail, the programmer should be aware that the keyboard and video also must support the local character set.

Message File Facility

In addition to the nationalization support provided by NLS and ENLS, you can use the message file facility to internationalize your application programs. Using this facility, you remove the messages from your application and place them in a separate message file. This eliminates linking the message strings with your program. As a result, your program code remains language-independent.

A message file actually exists in two forms: text and binary. You create your messages in text form. Then you convert this form to binary form so the messages can be accessed by your application. (See “Creating and Editing Message Files.”) Thereafter, your application can use the message operations in the standard operating system library to display the messages.

With each message file, the message file facility employs a message work area (MWA). The MWA, like the byte stream work area (BSWA) used by the Sequential Access Method, is an internal work area used by the facility to keep track of the messages in a file. (For details on the BSWA, see “Byte Stream” in the section entitled “Sequential Access Method.”)

Creating and Editing Message Files

To set up the message file for your program, use a text editor to open a text file. By convention, you give the file a name such as *PackageMsg.txt*. For each message string entry in the file, use the following format:

```
<Colon>Number<Colon><Delim>TextString<Delim>[,<Delim>
TextString<Delim>]
```

The delimiter can be any ASCII character. The portion of the syntax in brackets is optional and can be repeated as many times as you want. This allows a text string to be broken into any number of substrings. Each substring has a different delimiter, which can be useful if a message contains your delimiter character.

The following is a sample message:

:2000: "This is a sample text message."

This format is used by all standard message files.

To convert your text file to its binary form, use the Create Message File command through the Executive. (For details on Create Message File9, see the *CTOS Executive Reference Manual*.) Fill out the command form as shown below:

Create Message File	
[Text file]	<u>PackageMsg.txt</u>
[Message file]	<u></u>

By default, the name of the binary file is the same name as the text file name, except that the extension *.txt* is replaced with *.bin*. The name of the binary file created for the text file *PackageMsg.txt* is thus

PackageMsg.bin

Message File Operation Sets

There are actually three different sets of message file operations in the standard operating library. These are

- A standard set
- An alternate to the standard set
- A system service set

The standard and alternate sets are different versions of the same operations. They are generally used by applications using a large number of messages. The system service set is typically used by a program (such as a system service) that requires very few messages.

Standard Set

You use the standard set if your application only needs a single message file. These operations include `InitMsgFile`, `GetMsg`, and `PrintMsg`. An advantage to using these operations is that they do not require allocating memory for the MWA in your application.

Alternate Set

If, however, your application needs to use more than one message file at the same time, you would use operations in the alternate set. These operations can be easily identified by the characters *Alt* in their names, for example, `InitAltMsgFile` and `GetAltMsg`. For each message file your application uses, you must allocate memory in your application for a copy of the message work area (MWA). (See the description of the `InitAltMsgFile` operation in the *CTOS Procedural Interface Reference Manual* for details.)

System Service Set

There is a distinctly different set of message operations in the system service set. The set is used by system services or applications needing very few messages. The operations include `OpenServerMsgFile`, `CloseServerMsgFile`, and `GetServerMsg`. With these operations, the entire contents of the message file are copied into a memory buffer, and the messages are extracted from that buffer. These operations do not use macro expansion. (See “Using Macros With Messages.”)

Using Macros With Messages

Each message used with the standard or alternate set of message file operations may have one or more macros embedded in the text. A macro is identified by a leading percent sign (%), followed by one or more characters with no spaces. Macros provide added flexibility to the messages you create and are particularly useful when your application has a large number of messages.

At run time, the macros are expanded. Data for the macros can be supplied by using message file operations such as `ExpandLocalMsg`, `GetMsg`, `GetAltMsg`, `PrintMsg`, or `PrintAltMsg` or by programs calling such operations. The `GetMsgUnexpanded` operation (or its alternate version, `GetAltMsgUnexpanded`) can be used to retrieve a message from the message file and to place the message in memory without expanding any macros the message may contain.

(For details on defining message file macros, see Appendix F in the *CTOS Procedural Interface Reference Manual*.)

Native Language Support Operations

The native language support operations described below are arranged alphabetically in the categories NLS utility, ENLS utility, and message file. (See the *CTOS Procedural Interface Reference Manual* for a complete description of each operation.)

NLS Utility

GetNlsDateName

Returns a string containing the names of the months and the weekdays, as well as the strings: AM, PM, Noon, and Midnight.

GetNlsDateTimeTemplate

Returns a specified NLS date and time format template selected from the NLS Date and Time Formats table.

GetNlsKeycapText

Returns a string that contains the text to be displayed by programs when reference to a labeled key is desired.

GetNlsTable

Returns the specified NLS table in the memory area provided by the caller. The table is first looked up in the file *NLS.sys* loaded during operating system initialization. If not found, the NLS file described by *pbFileName/cbFileName* is opened, and the NLS table it contains is copied to the specified memory area.

GetPNlsTable

Returns the address of an NLS table in the NLS area specified by *pNlsTableArea*.

NlsCase

Translates a given character from lowercase to uppercase, or from uppercase to lowercase. It is recommended that you use *EnlsCase* instead of *NlsCase* for ease in nationalization.

NlsClass

Takes a given character and returns the class of that character. It is recommended that you use `EnlsCase` instead of `NlsCase` for ease in nationalization.

NlsCollate

Compares two strings to determine if they are equal or if one is greater than the other. Programs should use this operation rather than `NlsULCmpB` or `ULCmpB` for a richer set of collation rules.

NlsFormatDateTime

Converts from date/time format to textual string format. This operation employs a user-supplied format template in an alternate set of NLS tables linked with an application program (rather than the NLS tables loaded at boot time).

NlsGetYesNoStrings

Returns the strings defined in the yes and no strings table.

NlsGetYesNoStringSize

Returns the sizes of the strings defined in the yes and no strings table.

NlsNumberAndCurrency

Returns the address of the number and currency formats table.

NlsParseTime

Converts a string into an expanded date/time structure.

NlsSpecialCharacters

Returns the address of the special characters table.

NlsStdFormatDateTime

Converts from date/time format to text string format. This operation uses an index into a set of template strings in the Date and Time Formats table loaded at boot time. Programs should use this operation rather than `NlsFormatDateTime` or `FormatDateTime` for ease in nationalization.

NlsULCmpB

Functionality is the same as ULCmpB (described below).
NlsULCmpB is recommended for ease in nationalization.

NlsVerifySignatures

Validates an alternate set of NLS tables (that is, it ensures that the signatures embedded within the alternate table area provided match those expected to be there).

NlsYesNoOrBlank

Performs a case-insensitive string comparison against nationalized words meaning yes or no and also checks for a null string.

NlsYesOrNo

Performs a case-insensitive string comparison against nationalized words meaning yes or no.

OpenNlsFile

Opens the NLS file specified by *pbFileName/cbFileName*, loads the NLS tables contained in the file into short-lived memory, and returns the memory address of the tables.

ULCmpB

Compares two strings, using the lowercase to uppercase conversion table, if present, to carry out the case-insensitive comparison. ULCmpB returns 0FFFFh if the two strings are equal; otherwise, it returns a word containing the index of the first two characters in the strings that are different.

ENLS Utility

Character Processing

EnlsCase

Converts a given character from lowercase to uppercase or vice versa.

EnlsClass

Returns the class of a specified character.

EnlsGetChar

Returns a character from either the keyboard or an application-supplied memory area.

EnlsGetCharWidth

Returns the number of bytes in the character code.

EnlsGetPrevChar

Returns a character positioned to the immediate left of a specified offset within the specified string.

EnlsMapCharToStdValue

Maps a character code derived from the currently loaded character set to the corresponding U.S. standard character code.

EnlsMapStdValueToChar

Maps a U.S. standard character code to the corresponding character code in the currently loaded character set.

String Processing***EnlsAppendChar***

Appends a character to a character string and returns the length of the resulting string.

EnlsDeleteChar

Deletes a character from a character string and returns the length of the resulting string.

EnlsFieldEdit

Edits a string of characters on the video display. Input to the string is derived from characters typed at the keyboard. The characters typed can be keys defining edit commands or character data to be incorporated into the string. Editing is guided by an edit structure containing information such as the location and size of the string to be edited, the cursor position, whether or not to insert characters, and the exit character finishing the editing session.

EnlsFieldEditByChar

Edits a string of characters on the video display. Input to the string is derived from characters provided by the caller, one character at a time. The editing performed by this operation is guided by the same edit structure as the one used by *EnlsFieldEdit*.

EnlsFindC

Searches a string for a specified character and returns the string offset at which the character is found.

EnlsFindRC

Searches a string in reverse order for a specified character and returns the string offset at which the character is found.

EnlsInsertChar

Inserts a character into a character string. It moves over the other characters in the string array as many elements as necessary to provide space for inserting the character.

Line/Form Drawing

EnlsCbToCCols

Returns the number of screen or printer columns contained in a string of characters.

EnlsDrawBox

Draws a box to the video device.

EnlsDrawFormChars

Displays a string of line drawing form characters at the specified location.

EnlsDrawLine

Draws a line to the video device.

EnlsQueryBoxSize

Calculates the number of bytes required to save a box and its video information.

Message File

Standard Operations

The following operations are used by programs that require only a single message file:

CloseMsgFile

Closes an open message file.

ExpandLocalMsg

Expands any macro definitions contained in a message that resides in local memory.

GetMsg

Retrieves a message from the message file and places the expanded message in memory.

GetMsgUnexpanded

Retrieves a message from the message file and places the unexpanded message in memory (that is, it does not expand any macros that may be present in the message).

GetMsgUnexpandedLength

Returns the length of a message in its unexpanded form.

InitMsgFile

Opens a binary message file for subsequent retrieval of numbered messages.

PrintMsg

Retrieves a message from the message file and places the expanded message in a user-supplied video byte stream.

Alternate Operations

The following operations are used by programs that require more than one message file:

CloseAltMsgFile

Closes an open message file when any number of message files are open.

GetAltMsg

Retrieves a message from a message file opened by calling the `InitAltMsgFile` operation and places the expanded message in memory.

GetAltMsgUnexpanded

Retrieves a message from a message file opened by calling the `InitAltMsgFile` operation and places the unexpanded message in memory.

GetAltMsgUnexpandedLength

Returns the length of a message in its unexpanded form.

InitAltMsgFile

Opens a binary message file of the form *{Node}[VolName]<DirName>FileName* for subsequent retrieval of numbered messages by operations such as `PrintAltMsg` and `GetAltMsgUnexpanded`.

PrintAltMsg

Retrieves a message from the message file opened by calling the `InitAltMsgFile` operation and places the expanded message in a user-supplied byte stream.

System Service Operations

The following operations are used by programs, such as system services, that require few messages:

CloseServerMsgFile

Closes a message previously opened by a call to `OpenServerMsgFile`.

GetServerMsg

Extracts a particular message (string) from a message file that was previously initialized by `InitServerMsgFile`.

OpenServerMsgFile

Initializes a message file for use by a system service or an application program that is using a relatively small number of messages.

Appendix A

Operating System Features

In This Appendix

Table A-1 summarizes the features (or, in some cases, significant differences) supported on different versions of CTOS. The numbers in Table A-1 are associated with the comments in “Table A-1 Notes.” Table A-2 compares the CTOS memory management types.

For additional information, see the appropriate chapter in this manual or the documentation referenced in the table notes.

Table A-1. Operating System Features

	CTOS I 3.4 RMode	CTOS II 3.4 PMode	CTOS III 1.0	CTOS/XE 3.4 RMode	CTOS/XE 3.4 PMode
Disk Caching	no	yes	yes	yes ¹	yes
Change User Number System Requests²	yes	no	no	no	no
Code Sharing	no	yes	yes	no	yes
DMA Synchronous Communications	no	yes ³	yes ³	no	yes
Data Breakpoint⁴ Debugging	no	yes ⁵	yes	no	yes

continued

Table A-1. Operating System Features (cont.)

	CTOS I 3.4 RMode	CTOS II 3.4 PMode	CTOS III 1.0	CTOS/XE 3.4 RMode	CTOS/XE 3.4 PMode
Enhanced Video (EV) Color, 132- Column Mode	no	yes	yes	na	na
Extended Character Sets	yes	yes	yes	no	no
File System D Group Is Separate ⁶	no	yes	yes	yes	yes
I-Bus ⁷ Handlers Are Installable	yes	yes	yes	na	na
Installable Video Access Method (VAM) ⁷	no	yes	yes	yes	yes
Interrupt Vectors Are Sharable ⁸	no	yes	yes	no	no
Magnetic Card Reader	yes	yes	yes	no	no
Memory Disk	no	yes	yes	no	yes
Memory ⁹ Management Type	MP	VP	VM	MP	VP
Multi-Instance System Services ¹⁰	na	na	na	yes	yes

continued

Table A-1. Operating System Features (cont.)

	CTOS I 3.4 RMode	CTOS II 3.4 PMode	CTOS III 1.0	CTOS/XE 3.4 RMode	CTOS/XE 3.4 PMode
Protected Mode	no	yes	yes	no	yes
Relocatable When Swapped	no	yes	yes	no	yes
Remote Keyboard/ Video Service	no	yes	yes	yes	yes
Request Files Loadable	yes	yes	yes	yes	yes
SCSI Support	no	yes	yes	no	yes
Resource Access					
In Memory	no	no	yes	no	no
On Disk	yes	yes	yes	yes	yes
PM Support	no	no	yes	no	no

Table A-1 Notes

1. Only supported on disks with real mode SRP boards if a protected mode SRP board is present.
2. See the section entitled “System Services Management,” for details on ChangeUserName requests.
3. 386i workstations only.
4. See the *CTOS Debugger User’s Guide*.
5. 80386 and higher processors only.
6. See the *CTOS System Administration Guide*.
7. See your release documentation.
8. The same interrupt vector (level) can be shared by a trap vector and an interrupt handler.
9. The memory management types are as follows:
MP = Multipartition system.
VP = Variable-partition system.
VM = Virtual memory system.
10. See the section entitled “System Services Management.” This only applies to multiprocessor machines.

Table A-2. Summary of MP, VP, and VM Differences

Multipartition	Variable Partition	Virtual Memory
Code is never shared.	Code may be shared.	Code may be shared.
Code segments are written to the swap file repeatedly.	Code segments are written to the swap file only once.	Code segments read from (but never written to) run file.
Fixed partitions.	Optional variable partitions vary in size and location upon chain.	Partitions exist only for compatibility. Programs use as much virtual address space as they need.
Entire partitions are swapped.	Partitions may be partially swapped.	4K byte pages are faulted into physical memory frames by the paging service.
Partitions are created only by the program in the primary partition which is then forced to exit.	Partitions are created at any time without forced exit.	Partitions are created any time without forced exit.
Swapping is done by Context Manager.	Swapping is done by the operating system.	Paging is done by the paging service.
Swapping is only upon demand.	Swapping is upon demand and optionally by timer.	Paging is upon demand (by program execution, not user interaction).

Glossary

<\$> directories

An area on disk in which temporary files can be created. When a request with the directory name of <\$> is given as part of a file specification, the operating system expands the directory name to the form <\$000>nnnnn, where nnnnn is the user number associated with the application partition.

A

abort request

A system request issued to notify system services of a terminating client. Upon notification, the system service can release resources, such as open files and locked ISAM records, allocated to the terminating client. Issuing an abort request prevents requests from being returned to the client after it has been terminated and replaced in memory by another program.

accessed bit

See access rights byte.

access rights byte

One byte of a descriptor that contains information about a segment, such as whether the segment is present in memory, what the privilege level is, and whether the segment contains code or data. The segment descriptor access rights byte contains, in addition, an accessed bit for use by least-frequently-used algorithms in virtual memory management.

ACTION code

The keyboard code generated when a key (CANCEL, HELP, 0 through 9, or F1 through F10) is pressed in combination with ACTION. Programs can call ReadActionCode or ReadActionKbd to obtain the action code of a specified key combination. *See also* ACTION key.

ACTION key

A key processed specially, even in unencoded mode. The interpretation of all other keys is modified while **ACTION** is pressed. Key combinations that include **ACTION** are processed immediately when they are typed. Processing is independent of characters or keyboard codes stored in the type-ahead buffer. To allow a program to test for special operator intervention without preventing type ahead, key combinations including **ACTION** are available to applications for interpretation. *See also* **ACTION** code.

AL

Accumulator general register low byte.

allocation bit map

Controls the assignment of disk sectors. It consists of 1 bit for every sector on the disk. The bit is set if the sector is available. The allocation bit map is disk-resident.

allowed state

For a nonchord key, the chords that influence the character code generated. For a chord key, the allowed state determines whether a chord on a physical keyboard is allowed on a target keyboard.

alphanumeric style RAM

The video hardware controller for character attributes, such as blinking, half-bright, reverse video, and underlining.

alternate request procedural interface

A request issued by the caller for another user number. The alternate request procedural interface is constructed by prefixing the name of the operation the caller wants to make with **Alt** and specifying the user number as the first parameter to the operation. For example, to issue a **CloseFile** request on a specified file handle (fh) for user number 5, the request would be written as **AltCloseFile(5, fh)**.

application partition

A partition of user memory in which an application program can execute. A workstation can have any number of application partitions, with an application program executing concurrently in each. *See also* **system partition**.

application process

Executes code in the application program. It is not a system service process. *See also* system service process.

application profile

The keyboard an application expects is being used. *See also* system profile.

application program

Can consist of code, data, and one or more processes executing in an application partition. If the program is executing in a variable partition, the program's code can be located anywhere in memory and can be shared by the same type of program in a different variable partition.

application system control block (ASCB)

Communicates parameters, the termination code, and other information between an exiting application program and a succeeding application program in the same partition. *See also* variable length parameter block.

ASCB

See application system control block.

asynchronous mode

See asynchronous operation.

asynchronous operation

Asynchronous operation is a procedure or protocol that allows for a response within a window of time rather than at an exact time interval.

AVR

Automatic volume recognition.

B

bad sector file

Contains an entry for each unusable sector of a disk. The bad sector file is 1 sector in size.

backing store

A run file or swap file where pages reside when they are not in RAM.

base I/O address

An address on the X-Bus assigned to an X-Bus module by the bootstrap ROM. A base I/O address is used for I/O access to that module.

binary mode

One of three printing mode options in the printer: Generic Print System, pre-GPS spooler, and communications byte streams. Binary mode does not print the banner page before each file, send extra code not in the file to the printer, or recognize the escape sequence. *See also* image mode and normal mode.

bit-map workstation

Uses video software to emulate a character map to support character-only application programs. *See also* character map, character-map workstation, and video refresh.

block

An area of memory allocated for use by inter-CPU or cluster communications. *See also* X-block, Y-block, and Z-block.

blocked

A record file with several records per physical sector.

B-Net

A BTOS network consisting of nodes connected by communications lines over long distances. B-Net provides access to the system services of interconnected cluster configurations. *See also* CT-Net.

boot block

The area of memory that contains the information passed to the operating system by the bootstrap ROM.

bootstrap

To start (to boot) the system by reloading the operating system from disk. On other systems, this is often known as initial program load (IPL).

BP

Base pointer general register.

BSWA

See byte stream work area.

buffer management modes

The direct access method provides two modes of buffer management, write-through and write-behind.

buffer ownership table

Identifies processors currently using buffers on the local SRP board (shared resource processors only).

built-in

A program is built-in if it is part of the operating system core, which is always in memory. A dynamically installed program, on the other hand, is a program that can be added or removed at any time without regenerating the operating system. The file system is an example of a built-in system service; the Queue Manager service is dynamically installed.

BX

Base general register.

byte stream

A character-oriented, readable (input) or writable (output) sequence of 8 bit bytes used by the sequential access method (SAM) to transfer data to or from a device. *See also* byte stream work area, communications byte stream, disk byte stream, Generic Print System byte stream, keyboard byte stream, pre-GPS spooler byte stream, printer byte stream, sequential access method, tape byte stream, video byte stream, and X.25 byte stream.

byte stream work area

A 130 byte memory work area for the exclusive use of sequential access method procedures. Any number of byte streams can be open concurrently, using separate byte stream work areas.

C

cache hit

Indicates that the requested data buffer is in the cache.

cache miss

Indicates that the requested data buffer is not in the cache, but an alternate (stealable) buffer is available.

case value

A value used to identify which command invoked the current command when more than one possibility exists. The case value is held in the variable length parameter block and can be queried by a run file to determine which command actually invoked it.

cb

A variable prefix that indicates the count of bytes in a string of bytes.

change user number request

A system request issued (on multipartition operating systems only) to assign a new user number to a program that calls ConvertToSys. As a result of calling this request, user number 1 is maintained as the primary partition user number, and all existing system services are notified of the new user number for the program that converted to a system service.

character attribute

Controls the presentation of a single character. The standard character attributes are reverse video, blinking, half-bright, underlining, bold, and struck-through. Through system configuration parameters, you can substitute a color for the underlining, bold, and struck-through attributes on systems using standard VGA. *See also* screen attribute.

character cell

The pattern of illuminated dots (or pixels) on the video display of a workstation. The character cell size can be used by a program that calls the QueryVidHdw or QueryVideo operation to obtain other information describing the level of video capability of the workstation.

character code

The translation of the unencoded value and the chord state of a key. A character code can be 8, 16, or 32 bits long.

character map

The area of memory that holds the coded representation of the characters displayed on the video display of a character-map workstation. *See also* video refresh.

character-map workstation

Contains video hardware that supports the character map for the video display of characters. The hardware reads characters and attributes from memory and converts them from the extended ASCII (8 bit) representation to a pattern of dots (or pixels) that it displays on the video display of a workstation. During this translation, the video hardware references a font that is loaded into memory under program control. *See also* bit-map workstation and character map.

character mode

Mode in which an application reads keyboard events and is returned the displayable character code. *See also* unencoded mode.

character set

See standard character set.

check

A kernel primitive used by a client to determine if a message is queued at a specified exchange. If one or more messages are queued, the message that was first queued is removed from the queue, and its memory address is returned to the client. If no messages are queued, status code 14 ("No message available") is returned.

chord

A key that, when pressed, can influence the displayable character code generated by subsequent keys pressed, for example, **SHIFT**, **LOCK**, **CODE**, or **ALT**. Chords do not generate character codes in character mode.

chord state

A word that is a bit map of the chord keys currently pressed.

CLI

See command line interpreter.

CISR

See communications interrupt service routine.

client process

A process that requests a service provided by a system service. Any process, even a built-in operating system process, can be a client process, since any process can request system services. *See also* system service process.

cluster configuration

A local resource-sharing network consisting of a server connected to cluster workstations. One high-speed cluster communications channel is standard on each workstation. In cluster configurations connected to a server workstation, the server and all of the workstations connected to it use this channel for intercluster communications. For large clusters with a shared resource processor server, multiple communications channels are provided. The operating system executes in each cluster workstation and in the server. *See also* cluster workstation, B-Net, CT-Net, server, and server workstation.

cluster workstation

A workstation in a cluster configuration, connected to a server. *See also* cluster configuration and server.

Context Manager

A partition managing program. *See* partition managing program. (Also see your Context Manager manual.)

code segment

A variable-length (up to 64K bytes) logical entity consisting of reentrant code and containing one or more complete procedures. *See also* data segment, segment, and virtual code management facility.

color control structure

Used by programs to set the color in the color palette(s) and to turn the alpha character map and graphics bit map on or off independently. To set values for fields in the color control structure, the program must call the ProgramColorMapper operation. (For details on color programming, see the *CTOS Programming Guide*.) *See also* color palette, character map, and graphics bit map.

color mapper

A portion of the memory into which the color palette is loaded. The color mapper thus determines what colors are visible on the screen. *See also* color palette.

color palette

The color palette structure contains one or three palettes for programming alphanumeric and graphics color. (For details on color programming, see the *CTOS Programming Guide*.)

command descriptor block

A SCSI data structure used to communicate requests from an application program to a SCSI device.

command line interpreter

A software program on a shared resource processor that reads the job control language (initialization) file to install the processor's system services.

command name

The string a user types on the command line in the Executive to call a program. When the user presses RETURN, the Executive is given the command and responds by displaying the appropriate command form to the screen.

comm nub

The part of the operating system that dispatches RS-232-C communications interrupts. The comm nub passes control from the hardware interrupt to a user-written RS-232-C communications interrupt handler (also called an interrupt service routine) according to the instructions in an InitCommLine operation. When the interrupt handler has completed processing the interrupt, it passes control back to the comm nub.

common unallocated memory pool

A single contiguous area of memory in each application partition from which long-lived and short-lived memory segments are allocated.

communications byte stream

A byte stream that uses a communications channel. *See also* byte stream and byte stream work area.

communications interrupt service routine (CISR)

Similar to a mediated interrupt handler, except that a CISR serves only one of the two communications channels of the SIO communications controller (also called a communications interrupt handler). *See also* mediated interrupt handler.

communications status buffer

A system structure that contains statistics for the server and the workstations connected to it.

configuration file

Specifies data to be used by the operating system, a utility, or an application program. Example configuration files are *Config.sys* and the device configuration files created by the **Create Configuration File** command through the Executive.

conforming code/expand-down data segment bit

One of the bits in the access rights byte. *See* access rights byte.

context switch

Saving the register contents in the process control block (PCB) of a process that is interrupted (preempted by a process with a higher priority). When the interrupted process is rescheduled for execution, the operating system restores the contents of the registers. A context switch permits an interrupted process to resume execution as though it were never interrupted. *See also* process, process context, process control block, and task switch.

control information

The data after the request block header and before the first request address/size (pb/cb) pair.

CP

Cluster processor.

CPU

The CPU (central processing unit) is the microprocessor.

crash dump area

The crash dump area (the file *[Sys]<Sys>CrashDump.sys*) contains a binary memory dump in the event of a system failure.

CRC

cyclic redundancy check.

CS

Code segment.

current

A current user number is the one that is presently executing.

CT-Net

A CTOS network consisting of nodes connected by communications lines over long distances. A node is a junction in CT-Net (a server workstation or a processor board on a shared resource processor). CT-Net provides access to the system services of interconnected cluster configurations. *See also* B-Net.

CTOS

Unisys operating systems, which run on 80X86 microprocessors.

cursor RAM

Part of the advanced video capability, which allows software to specify a 10 by 15 bit array as a pattern of pixels in place of the standard cursor (a blinking underline). The cursor bit array is superimposed on the character and blinks.

D

DAM

See direct access method.

data segment

Contains data; it can also contain code, although this is not recommended. If a data segment is shared among processes, concurrency control is the responsibility of those processes. A data segment that is automatically loaded into memory when its containing run file is loaded is called a static data segment, to differentiate it from a dynamic data segment that is allocated by a request from the executing process to the memory management facility. *See also* code segment and segment.

date/time format

Provides a compact representation of the date and the time of day that precludes invalid dates and allows simple subtraction to compute the interval between two dates. The date/time format is represented in 32 bits to an accuracy of 1 second.

DAWA

See direct access work area.

DCB

See device control block.

decoding value(s)

An ordered set of unencoded values that will produce a character code.

default response exchange

A unique response exchange assigned to a process when that process is created. This exchange is automatically used as the response exchange whenever the client process uses the request procedural interface to a system service. Use of the default response exchange is limited to synchronous requests (that is, requests for which the client must wait for a response before issuing them again). *See also* exchange and response exchange.

descriptor privilege level

A feature of protected mode that indicates the privilege level of a segment. *See also* access rights byte.

device

A physical hardware entity. Printers, tape, floppy disks, and hard disks are examples of devices.

device control block (DCB)

A memory-resident control block for a physical device. The DCB contains information about the device generated at system build. For a physically addressed disk (*see* physical address device), the information includes the number of tracks on the disk, the number of sectors per track, and so forth. The DCB contains the memory address of a chain of I/O blocks.

device-dependent

Describes program interfaces closest to the actual hardware. A device dependent program performs I/O to a limited number of devices. *See also* device-independent.

device-independent

Describes program interfaces that are not close to the hardware. A device-independent program can perform I/O to a variety of devices. The sequential access method operations, such as `OpenByteStream`, `ReadByteStream`, and `CloseByteStream`, are device-independent operations. *See also* device-dependent.

device password

Protects a device.

device specification

Consists of a device name. The device name is the only element of a device specification.

diacritical key handling

Displaying characters with diacritical marks, such as the German **a** with an umlaut.

diacritics

Keys working in pairs that produce a diacritical result such as a German **a** with an umlaut. Pressing the first key enables diacritical mode (nothing is returned); the second key returns the result.

direct access method

Provides random access to disk file records identified by record number. The record size is specified when the DAM file is created. DAM supports COBOL relative I/O, but can also be called directly from any of the Unisys languages. *See also* direct access work area.

direct access work area (DAWA)

A 64 byte memory work area for the exclusive use of the direct access method procedures. Any number of DAM files can be open simultaneously using separate DAWAs. *See also* direct access method.

direct memory access (DMA)

Access to memory that does not require processor intervention. A DMA controller in the processor module controls the transfer of data over the X-Bus from a memory master or master/slave to the main processor's memory.

direct printing

Transfers text directly from application program partition memory to the specified parallel or serial printer interface of the workstation on which the application program is executing. Direct printing is always accessed through the sequential access method (disk byte streams). *See also* disk byte stream, spooled printing, and pre-GPS spooler byte stream.

directory

A collection of related files on one volume. A directory is protected by a directory password.

directory password

Protects a directory on a volume.

directory specification

Consists of a node name, volume name, and a directory name.

dirty

Modified in memory but not yet written to secondary storage.

disk byte stream

A disk byte stream is a byte stream that uses a file on disk. *See also* byte stream and byte stream work area.

disk controlling processor

See master processor.

disk extent

One or more contiguous disk sectors that compose all or part of a file.

disk partition

A disk structure organizing the disk into regions, each of which can accomodate a separate operating system environment.

DMA

See direct memory access.

doorbell interrupt

A handshake protocol in which a device interrupts another device by writing to a doorbell interrupt location. The device being interrupted responds by servicing the interrupt and resetting the interrupt request on the device generating the interrupt. A timeout may or may not be implemented. Doorbell interrupts are used on shared resource processors for notifying a processor board that it has received a message from a remote processor board.

DP

Data processor.

DS

Data segment.

DTE

Data terminal equipment.

dynamic data segment

See data segment.

dynamically installed system service

A program that was loaded as an application program and converted itself into a system service using the ConvertToSys operation. Once installed, a dynamically installed system service has the same capabilities as a system service linked with the system image during system build. A dynamically installed system service uses CTOS operations (rather than system build parameters) to identify the request codes that it serves, specify its execution priority, establish its interrupt handlers, and so forth.

E

EAR

See extended address register.

electrical layer

Combined with the physical layer of the SCSI standard, defines the form and shape of the SCSI connectors, the voltage levels, and the timing relationships of the electrical signals used.

emulating

Mapping the chord state and unencoded values on the source keyboard hardware to the chord state and unencoded values necessary to produce the same character on the target keyboard.

ENLS

See extended native language support.

EOF

End of file.

EOI

End of interrupt.

EOM

End of medium.

EOT

End of tape.

erc

A one word status code returned by a function call.

ES

Extra segment.

escape sequence

A special sequence of characters that invokes special functions. *See also* spooler escape sequence, submit file escape sequence, and multibyte escape sequence.

event

In the context of timer management, an event occurs when an interval elapses. *See also* system event.

event-driven priority-ordered scheduling

When processes are scheduled for execution based on their priorities and system events, not on a time limit imposed by the scheduler. *See also* process and system event.

exchange

The path over which messages are communicated from process to process (or from interrupt handler to process). An exchange consists of two first-in, first-out (FIFO) queues: one of processes waiting for messages and the other of messages for which no process has yet waited. An exchange is referred to by a unique 16 bit integer. *See also* default response exchange and response exchange.

Executive

An interactive application program that accepts commands from the workstation user and requests the operating system to load programs to execute those commands. This function can be performed by the Unisys Executive or by a user-written Executive. The Executive is loaded from the file *[Sys]<Sys>Exec.run* if specified as the *SignOnExitFile*. The file *[Sys]<Sys>Exec.run* usually contains the Unisys Executive; however, it can contain a user-written Executive.

exit run file

A user-specified file that is loaded and activated when an application program exits. Each application partition has its own exit run file.

expand-down segment

A segment that can be increased in size by changing the lower limit to a lower memory address. The segment offset is greater than the limit.

expand-up segment

A segment that can be increased in size by changing the upper limit to a higher memory address. The segment offset is less than or equal to the limit.

export

A procedure in a DLL, or exporter, that may be directly accessed by a client.

extended character set

A character set containing more than 256 characters. Characters need to be defined by more than one byte.

extended native language support

Operations that process single byte or multibyte characters.

extended partition descriptor

Located in each application partition and contains specifications for the current application file and exit run file.

extended system service

A system service that is dynamically installable. The extended system services are described in the *CTOS Programming Guide, Volume II*.

extended user control block

Located in each application partition and contains the offset of the partition descriptor. *See also* partition descriptor.

extension file header blocks

A volume control structure for a file containing more than 32 disk extents. *See also* file header block.

external interrupt

Caused by conditions that are external to the processor and are asynchronous to the execution of processor instructions. There are two kinds of external interrupts: maskable and nonmaskable. *See also* internal interrupt, maskable interrupt, and nonmaskable interrupt.

F

FAB

See file area block.

FALSE

Represented in a flag variable as 0.

far procedure

Referenced by the procedure's code segment (CS) and offset (IP). A far procedure can be called by procedures within the same or from within a different module.

FCB

See file control block.

fh

File handle.

FHB

See file header block.

FIFO

First-in, first-out.

file

A set of related bytes (on disk) treated as a unit.

file area block

A memory-resident system structure corresponding to a disk extent in an open file. The file area block (FAB) specifies where the sectors are and how many there are in the disk extent. The FAB is pointed to by a file control block or another FAB. *See also* disk extent.

file control block

A memory-resident system structure corresponding to an open file. The file control block (FCB) contains information about the file such as the device on which it is located, the user count (that is, how many file handles currently refer to this file), and the file mode (modify, peek, or read). The FCB is pointed to by a user control block and contains a pointer to a chain of file area blocks.

file handle

A 16 bit integer that, in combination with a user number, uniquely identifies an open file. It is returned by the OpenFile operation and is used to refer to the file in subsequent operations such as Read, Write, and DeleteFile.

file header block

A disk-resident volume control structure corresponding to a file. The file header block (FHB) of a file contains information about that file such as its name, password, protection level, the date/time it was created, the date/time it was last modified, and the disk address and size of each of its disk extents. *See also* extension file header block.

file password

Protects a file in a directory on a volume.

file protection level

Specifies the access allowed to a file when the accessing process does not present a valid volume or directory password.

filter process (user-defined)

A user-written system service process that can be included in the system image at system build or dynamically installed at any time. A filter process is interposed between a client process and a system service process that operate as though they are communicating directly with each other. The service exchange table is adjusted to route requests through the desired filter process.

filter process (local file system)

See local file system.

fixed partition

Always uses a fixed amount of memory. *See also* variable partition.

font

A bit array for each of the 256 characters in the character set that defines the representation of each character when displayed on the video display.

font RAM

For the video, contains a bit array for each of the 256 characters in the character set. The font RAM can be modified under software control.

ForwardRequest

A kernel primitive that can be used by a one-way pass-through filter to forward a request block to a system service for further processing. The system service responds directly to the client.

FP

File processor.

frame

(1) A separate, rectangular area of the screen. A frame can have any desired width and height (up to those of the entire screen). (2) A 4K byte region of an application that is faulted into memory as the code it contains is requested.

frame descriptor

A component of the video control block (VCB) containing all the information about one of the frames.

free memory

Unused system memory.

full file specification

Consists of a node name, volume name, directory name, and file name.

G

GDT

See global descriptor table.

Generic Print System (GPS)

A set of dynamically installed system services, which work together to handle communication between application programs, the operating system, and the printers and plotters currently installed. GPS is the software underlying the Print Manager. (For details, see the *Generic Print System Programming Guide*.)

Generic Print Access Method (GPAM)

An object module library providing high-level, device-independent formatting commands for printing complex documents (containing text, graphics, and/or special text attributes). (For details, see the *Generic Print System Programming Guide*.)

Generic Print System byte stream

A byte stream sent to a GPS printing device. See also byte stream, byte stream work area, Generic Print System, and Print Manager.

global descriptor table (GDT)

A protected mode structure that contains descriptors for segments, which are shared by all programs. See also local descriptor table (LDT) and segment descriptor.

global initialization

DLL initialization procedures called once when the first DLL client is loaded.

global linear address

A linear address valid for all users.

global request

A request that can be served by a request-based system service on any SRP processor board. *See also* local request.

global segment

A DLL segment, a single copy of which is shared by all clients.

GP

General processor.

GPS

See Generic Print System.

graphics bit map

The graphics bit map is a multiplane bit map that is manipulated by operations in the graphics library. (See the *Graphics Programming Guide*.)

graphics style RAM

Video hardware controller of character attributes, color, and intensity on color graphics workstations.

H

hashing techniques

See randomization techniques.

heap

Dynamic memory consisting of segments that can be allocated and deallocated in any order.

high-level interface

Programmatic interfaces, which, when used exclusively, provide device-independence to a program.

high-resolution

Video resolution of a graphics controller that produces 12 X 20 pixel (illuminated dot) characters on the screen.

host adapter

A SCSI hardware device interposed between a computer system and a SCSI bus. The device usually performs the lower layers of the SCSI protocol and normally operates in the initiator role.

huge segment

A segment greater than 64K bytes.

hypersegment

An application partition component. The hypersegments are the local descriptor table, user structure, program code, and dynamically allocatable long-lived and short-lived memory.

I

I-Bus style

Keyboards using the I-Bus protocol. Most CTOS keyboards are in this class and include the model K1 through K5 keyboard and the SuperGen I-Bus keyboard (SG101-K).

I-key

A keyboard value used to index various keyboard data block tables to locate information about the key. The I-key may be a key post value generated by the hardware, or it may be an emulated value.

ICC

See inter-CPU communication.

ICMS

See Intercontext Message Server.

IDT

See interrupt descriptor table.

image mode

One of three printing options in the Generic Print System, pre-GPS spooler, printer, and communications byte streams. Image mode prints the banner page before each file and recognizes escape sequences but performs no code conversions. *See also* normal mode and binary mode.

import

A Dll procedure called in a client program, or importer.

import library

A library containing the names of DLLs and DLL procedures to be called by a client. The loader uses the client's import library to resolve external references to DLL procedures in the client program.

initiator

A SCSI device (usually a host computer system) that requests an I/O process to be performed by another SCSI device (target).

instance initialization

DLL initialization procedures called each time a new client first references the DLL.

instance segment

A DLL segment, an individual copy of which is made for each client.

interboard routing

Passing request and response messages between processor boards of a shared resource processor.

Intercontext Message Server (ICMS)

Used by application programs to communicate with programs in other application partitions. The requesting program sends an interprocess communication message to ICMS. ICMS, in turn, uses interprocess communication to forward the message to the receiving program. ICMS prevents messages from being sent to programs while they are swapped out of memory.

Indexed Sequential Access Method (ISAM)

Provides efficient, yet flexible, random access to fixed-length records identified by multiple keys stored in disk files.

input byte stream

See byte stream.

interactive

A program is that interfaces with the user. The Executive is an example of an interactive command interpreter.

inter-CPU communication

Communication used by the kernel for routing messages between SRP processor boards.

interface level

Implies the relative degree of program control over a hardware device.

internal interrupt

Caused by and is synchronous with the execution of processor instructions.

internationalizing

Making software localizable without having to alter source code.

interrupt

External or internal; an event that interrupts the sequential execution of processor instructions. When an interrupt occurs, the current hardware context (the state of the hardware registers) is saved. This context save is performed partly by the processor and partly by the operating system. *See also* external interrupt, internal interrupt, maskable interrupt, non-maskable interrupt, and pseudointerrupt.

interrupt descriptor table

The protected mode equivalent of the interrupt vector table. The tables function similarly in that each directs interrupts to the appropriate interrupt handling routines.

interrupt handler

Code that gains control when an interrupt occurs. Since an interrupt handler is not a process, it can only invoke only a few specific operations. Operating system interrupt handlers are provided for each interrupt type.

Interrupt number

Each potential source of interrupt is assigned an interrupt number in the range 0 to 255 that identifies the interrupt type (source of the interrupt). When an interrupt occurs, the hardware recognizes the interrupt type and the applicable interrupt number. The processor uses this number to vector the interrupt to the appropriate interrupt handler. *See also* interrupt and interrupt handler.

Interrupt service routine

An interrupt service routine is an interrupt handler. *See* interrupt handler.

Interrupt vector table

The interrupt vector table is a real mode structure that contains a 4 byte entry for each interrupt type. Each 4 byte entry contains the logical memory address (CS:IP) of the first instruction to be executed when an interrupt of that type occurs.

IOB

See I/O block.

I/O Block (IOB)

Used by the operating system as temporary storage during Read, Write, and other I/O operations. The IOB contains information obtained from the request block. The number of IOBs specified at system build must be adequate for the maximum number of I/O operations that will be in progress simultaneously. The IOB is memory-resident.

IPC

Interprocess communication.

IPL

Initial program load.

ISAM

See Indexed Sequential Access Method.

K

kernel

The most primitive and the most powerful component of the operating system. It executes with a higher priority than any process but it is not itself a process. The kernel is responsible for the scheduling of process execution; it also provides IPC primitives.

keyboard byte stream

A byte stream that uses the keyboard. *See also* byte stream and byte stream work area.

keyboard code

(For I-Bus style keyboards) an 8-bit value describing the keyboard event. When the high-order bit is 0, the 7 low-order bits indicate the hardware key post downstroke. When the high-order bit is set, the value indicates the key post upstroke.

keyboard data block

The smallest complete set of data supporting translation or emulation. Although keyboard data blocks are alternately called *translation* or *emulation tables*, they contain a variety of structural components (including translation or emulation subtables) that support the translation or emulation process. The file *NlsKbd.sys* is comprised of keyboard data blocks.

L

language definition

Unique language requirements, such as its currency symbols and date/time formats.

LBA

Logical block address. *See also* logical block address (LBA) device.

LDT

See local descriptor table.

lfa

See logical file address.

limit checking

A protection feature of protected mode that places limitations on the memory a program can access.

linear memory address

Linear memory addresses are relative distances from memory address 0 in physical memory and can be compared to each other on this basis. See also linear address space.

linear address space

Begins at physical memory address 0 and extends linearly to the maximum amount of physical memory that actually can be addressed by a program.

link block

A system data structure that is used to queue messages at exchanges. Each link block contains the address of the message and the address of the next link block (if any) that is linked onto the exchange. Two pools of link blocks are specified at system build, a general pool and a special pool used only by the PSend primitive. A call to the Request primitive reserves 1 link block from the general pool for the corresponding Respond primitive. For these reasons, the number of link blocks in each pool can be specified at system build.

Linker

Links one or more object files into a run file to be loaded into memory. (For details on the Linker, see the *CTOS Programming Utilities Reference: Building Applications*.)

loadable request file

A file containing request definitions for a request-based system service(s). The loadable request file is used to merge new requests with the requests already defined in the system request file, *Request.sys*. The merge occurs during installation of the system service(s) onto the system disk. When bootstrapped, the operating system reads *Request.sys*, loads it into memory, and adds the requests it contains to the basic request routing table.

local descriptor table (LDT)

A protected mode structure in memory that contains descriptors for segments accessible to a run file. The operating system constructs the LDT based on information provided by the Linker.

local file system

Allows a cluster workstation to access files on a local hard disk(s) as well as files on the hard disk(s) at the server. The filter process of the local file system intercepts each file access request and directs it to the local file system or to the server workstation.

local linear address

A linear address valid for a single user.

local resource-sharing network

A cluster configuration consisting of cluster workstations connected to a server.

local request

Served by a system service on the same processor board of a shared resource processor as the client.

localizing

Making software reflect a language definition.

locked

Not subject to being removed from physical memory (for example, a locked cache entry or a locked page).

log file

An error-logging file. An entry is placed in the system log file (*/Sys/<Sys>Log.sys*) for each recoverable and nonrecoverable device error. This file can be used as a general-purpose logging file, for example, to write entries for accounting information for system services.

logical block address (LBA) device

A type of device that typically contains controller intelligence, hiding the physical details of the disk organization and presenting the appearance of a contiguous address space from block zero up to the maximum capacity of the disk. This type of device is in contrast to a physical address (PA) device.

logical file address

Specifies a byte position within a file relative to the start of the file (byte 0). A logical file address (lfa) is a 32 bit unsigned integer that must be on a sector boundary and is therefore a multiple of 512 bytes.

logical memory address

The 32 bit memory address (usually abbreviated as memory address) as viewed by the application program. It consists of a segment address (SA) and a relative address (RA) or offset.

logical session layer

A layer of the SCSI standard that defines the meanings of commands sent to SCSI devices, the actions at the devices as a result of the commands, and the data (transferred to or from the devices) accompanying the commands.

logical unit

A physical or virtual peripheral device addressable through a SCSI target.

logical unit number

An encoded 3 bit identifier of a logical unit.

long-lived memory

Memory persisting across chain to another program in the partition.

low-level interface

A programmatic interface that is close to the actual hardware. Programs using low-level interfaces are device-dependent.

low resolution

The video resolution of a graphics controller that produces 9 X 12 pixel (illuminated dot) characters on the screen.

M

mapped unencoded mode

A submode of unencoded mode in which the key values returned are mapped to the corresponding unencoded K1 keyboard codes shown in Table C-1 in the *CTOS Procedural Interface Reference Manual*.

maskable interrupt

Given a priority and controlled by the programmable interrupt controller (PIC) and can be masked (ignored) by the use of the processor interrupt-enable flag. A maskable interrupt can be masked selectively by programming the PIC. *See also* external interrupt and nonmaskable interrupt.

master agent

Reconverts a message from a workstation connected to the cluster line to an interprocess request that it queues at the exchange of the request-based system service on the server that actually performs the intended function. The master agent includes the cluster code at the server that polls the cluster workstation for requests. *See also* workstation agent.

master processor

A processor with a file system in a shared resource processor system that bootstraps first then boots all the other processor boards in the system. The master processor is connected to the front keyswitch panel and is the leftmost processor in the primary cabinet. A master processor is sometimes called a disk controlling processor and can be a GP+SI, a DP, or an FP.

master file directory

A disk-resident volume control structure containing an entry for each directory on the volume, including the *Sys* directory. The entry contains the directory name, password, location, and size.

mediated interrupt handler (MIH)

One of two procedural styles for handling an interrupt. The other style is a raw interrupt handler. When compared to a raw handler, a mediated interrupt handler executes more slowly. This is because it can be written in a high level language, interrupts are enabled during its execution (so that it can be pre-empted), and it can communicate its results to processes through certain kernel primitives. *See also* interrupt handler and raw interrupt handler.

memory address

See logical memory address.

message

The entity transmitted between processes by the interprocess communication facility. It conveys information and provides synchronization between processes. Although only a single 4 byte data item is literally communicated between processes, this data item is usually the memory address of a larger data structure. The larger data structure is called the message, while the 4 byte data item is conventionally called the address of the message. The message can be in any part of memory that is under the control of the sending process. By convention, control of the memory that contains the message is passed along with the message.

MFD

See master file directory.

MIH

See mediated interrupt handler.

model of segmentation

A model to which a compiler can make code conform.

modify mode

One of three ways that a file can be opened using an operation, such as `OpenFile` or `OpenFileLL`, that can open a file. Modify mode is used to write to the file. Access in modify mode permits the returned file handle to be used as an argument to all operations that expect a file handle. *See also* peek mode and read mode.

module definition file

A file that defines the specific requirements of a DLL or DLL client module to the Linker.

multibyte character

A character defined by more than one byte.

multibyte escape sequence

A special sequence of characters that is available to disable video byte stream interpretation of special characters except 0FFh.

multiprogramming

The ability to run more than one application in memory at the same time. Multiprogramming supports the independent invocation and scheduling of multiple processes. In addition, it provides for concurrent I/O and for multiple processor implementations.

multitasking

See multiprocessing.

multiprocessing

The ability for any program to have more than one process (thread of execution). Multiprocessing also is called multitasking.

N

nationalization

Internationalizing and localizing software.

near procedure

Referenced by the offset (IP) of the procedure's memory address. Near procedures can be called only by other procedures within the same module.

Net Agent

Receives requests over the network destined for request-based system services located at remote nodes and forwards these requests to the remote nodes. *See also* B-Net, CT-Net, and Net Server.

Net Server

Responds to requests from Net Agents. The Net Server receives a request block from the Net Agent, executes the request on behalf of the remote client, and returns the response to the originating Net Agent. *See also* Net Agent.

network

See B-Net and CT-Net.

NLS table

One of several (optional) internationalizable tables supplied as part of Standard Software in the source file, *[Sys]<Sys>Nls.asm*. The NLS tables can be edited, assembled, and linked to create the NLS configuration file *[Sys]<Sys>Nls.sys*.

NMI

See nonmaskable interrupt.

node

The first element (node name) of a full file specification. Node also refers to a server in a network. *See also* B-Net and CT-Net.

nonmaskable interrupt (NMI)

An interrupt with a higher priority than a maskable interrupt. An NMI cannot be masked through the use of the processor interrupt-enable flag; however, bits in the I/O control register allow each of the four conditions that cause NMI to be masked individually. These conditions are write-protect violation, nonexistent or device-addressed memory parity error, and power failure detection. *See also* maskable interrupt.

nonoverlapped

In the context of the structured file access methods, nonoverlapped means that a call to a read or write operation does not return to the calling program until all associated input or output is complete.

normal mode

One of three printing options in the Generic Print System, printer, pre-GPS spooler, and communications byte streams. Normal mode prints the banner page before each file, converts tabs into spaces and end-of-line characters to device-dependent codes, and recognizes the escape sequences for manual intervention. *See also* binary mode and image mode.

null process

A process that is always ready to run. It has priority 255, which is lower than any real process. It exists only to simplify the operating system scheduler algorithm.

O

object module procedure

A procedure supplied as part of an object module file. It is statically linked with the user-written object modules of an application program and is not supplied as part of the system image. Most application programs only require a subset of these procedures. When the application program is linked, the desired procedures are linked together in the application run file.

offset

Also called the relative address; the distance, in bytes, of the target location from the beginning of the segment.

operation

An operating system kernel primitive, system service, system-common procedure, or object module procedure.

OS

Operating system.

output byte stream

See byte stream.

overlapped

In the context of structured file access methods, overlapped means that although the application program makes a call to a read or write operation and that operation returns, I/O can continue overlapped automatically with the computations of the application program.

overlay area

A buffer in the memory of an application partition used to contain all nonresident code segments. This buffer area must be large enough to contain the largest nonresident code segment. A larger buffer permits more code segments to be kept in the main memory of the partition and improves system performance. *See also* virtual code management facility.

overlay program

A program that uses the virtual code management facility. *See* virtual code management facility.

oversubscribed

Physical memory condition in which the working set of all programs in the system exceeds the size of physical memory.

P**PC-style**

A keyboard (used with the SuperGen Series 5000, for example) that does not use I-Bus protocol (SG102-K). Other documentation may refer to this keyboard as the *PS/2-style* keyboard.

page

A 4K byte section of a program in the linear address space.

page cleaning

Writing the contents of a modified (dirty) page to backing store.

paragraph

A 16 byte memory area the physical memory address of which is a multiple of 16. In real mode, segment addresses are paragraph-aligned.

partition configuration block

Located in each application partition and contains the offsets of the application system control block, batch control block, and extended partition descriptor. *See also* application system control block, batch control block, and extended partition descriptor.

partition keyboard information structure

Structure containing keyboard data (for example the chord state) for the currently executing user number.

partition descriptor

Located in each application partition and contains the partition name, the boundaries of the partition and of its long-lived and short-lived memory areas, and internal links to partition descriptors in other partitions.

partition handle

Another name for a user number. *See* user number.

partition management facility

Permits concurrent execution of multiple application programs, each in its own partition. It provides operations for creating, managing, and removing application partitions. *See also* partition managing program.

partition managing program

A partition managing program such as Context Manager uses the partition management operations described in the section entitled "Partitions and Partition Management" to manage multiple application partitions in memory. Each partition can contain an executing application program.

pb

A variable prefix meaning the memory address of a string of bytes.

pb/cb

A 6 byte entity consisting of the 4 byte memory address of a byte string followed by the 2 byte count of the bytes in that byte string.

PCB

See process control block.

peek mode

One of three ways that a file can be opened using an operation, such as `OpenFile` or `OpenFileLL`. If the file is currently open in peek mode, access in either read or modify mode is permitted. If, however, the program that originally opened the file in peek mode attempts to read or to modify the file after it has been modified by another program, status code 210 (“Bad file handle”) is returned. *See also* modify mode and read mode.

physical address space

The actual amount of memory that can be addressed by a program. In real mode, the physical address space is 1 megabyte. In protected mode, the physical address space is determined by the system processor and its hardware limitations.

physical address (PA) device

A direct-access device of an earlier generation that did not have “intelligence” embedded within the disk controller. The volume home block (VHB) had to explicitly describe the size and geometry of the disk platters in terms of cylinders per disk, tracks (heads) per cylinder, physical sectors per track, and bytes per physical sector. This type of device is in contrast to a logical block address (LBA) device that does have embedded intelligence.

physical layer

Combined with the electrical layer of the SCSI standard, defines the form and shape of the SCSI connectors, the voltage levels, and the timing relationships of the electrical signals used.

physical memory address

Location of a byte of volatile memory.

physical record

In the context of file access methods, an entity that consists of the record header, the record data, and the record trailer stored in contiguous bytes.

PIC

See programmable interrupt controller.

PISR

Printer interrupt service routine.

PIT

See programmable interval timer.

prefaulting pages

Loading pages into frames when the pages have not been accessed. Prefaulted pages are contiguous with a page that is accessed and must be faulted in.

pre-GPS spooler byte stream

A pre-GPS spooler byte stream automatically creates a uniquely named disk file for temporary text storage. It then transfers the text to the disk file and expands the disk file as necessary. When the pre-GPS spooler byte stream is closed, a request is queued to the spooler to print the disk file and delete it after it is printed. This is spooled printing. *See also* byte stream and byte stream work area.

present bit

A bit (in protected mode) in a descriptor that indicates whether a segment is in memory. *See also* descriptor.

primary partition

When a single application partition exists in memory, this partition is called the primary partition. A primary partition is not under the control of a partition managing program, such as Context Manager.

primary task

The first run file that is loaded into an application partition containing more than one run file. The primary task can be loaded with the LoadPrimaryTask operation by a partition managing program in a different partition or with a Chain, Exit, or ErrorExit operation by a program in the same partition. The primary task in turn loads additional tasks, called secondary tasks, in its own partition with the LoadTask operation.

primitive

An operation performed by the kernel. *See also* kernel.

printer byte stream

A byte stream that performs direct printing. It can use either a Centronics-compatible printer connected to a parallel printer port or an RS-232-C-compatible printer connected to communications Channel A or B of the workstation on which the application program is executing. *See also* byte stream and byte stream work area.

printing mode

See binary mode, image mode, and normal mode.

priority

Indicates a process's importance relative to other processes and is assigned at process creation. Priorities range from a high of 0 to a low of 254.

procedure

A subroutine.

process

An independent thread of execution for a program along with the context (that is, the processor registers) necessary to that thread. One or more processes are created each time a program is scheduled for execution. A process is assigned a priority when it is created so that the operating system can schedule its execution appropriately. *See also* priority.

process context

The collection of all information about a process. The context has both hardware and software components. The hardware context of a process consists of values to be loaded into process or registers when the process is scheduled for execution. This includes the registers that control the location of the process's stack. The software context of a process consists of its default response exchange and the priority at which it is to be scheduled for execution.

process control block (PCB)

A system data structure that is the root of the combined hardware and software context of a process. A PCB is the physical representation of a process. *See also* process context.

processor

Consists of the central processing unit (CPU), memory, and associated circuitry. The hardware manuals refer to the processor as the mainframe. *See also* CPU.

processor ordinal number

Is the processor's position (starting with 0) within the shared resource processor. (For an illustration of processor ordinal numbering, see "Understanding Hardware" in the *CTOS System Administration Guide*.)

program

Consists of executable code, data, and one or more processes. A program is created by translating source programs into object modules and then linking them together. This results in a run file that is stored on disk. When requested by a currently active program, such as the Executive, the operating system reads the run file into the application partition, relocates intersegment references, and schedules it for execution. The new run file can coexist with or replace other run files. *See also* primary task, run file, and secondary task.

programmable interrupt controller (PIC)

Controls when certain external hardware interrupt types can occur. Each interrupt type can be connected (wire OR'd) to one or more device controllers, such as a keyboard controller or an RS-232-C communications controller. In some hardware configurations, however, a master PIC can be connected to a slave PIC. The slave PIC multiplies the number of external interrupt sources that can be uniquely identified and priority ordered. *See also* interrupt and interrupt levels. (For details, see the hardware manuals for the various processors.)

programmable interval timer (PIT)

Provides high-resolution, low-overhead activation of user pseudointerrupt handlers.

protected mode

One of the CTOS operational modes. In protected mode, application programs can use all available free memory up to the maximum allowed by the processor and the hardware.

pseudointerrupt

Implemented in software rather than in hardware and, in this sense, is not really an interrupt. However, a pseudointerrupt causes an interrupt handler to be executed as a real interrupt is and has the same responsibilities and privileges. *See also* interrupt.

Q**Queue Manager**

A system service that controls named, priority-ordered, disk-based queues. For details on the Queue Manager, see the *CTOS Programming Guide, Volume II*.

R**RA**

See offset.

RAM

Random access memory.

raw interrupt handler (RIH)

A procedure that executes quickly in handling an interrupt. An RIH is useful for servicing a high-speed, non-DMA device that causes frequent interrupts. *See also* interrupt handler and mediated interrupt handler.

raw unencoded mode

Mode in which the application is returned the unencoded value of key returned from the attached hardware.

RCB

Request control block.

read mode

Read mode is one of three ways that a file can be opened using an operation, such as `OpenFile` or `OpenFileLL`. Access in Read mode permits the returned file handle to be used as an argument only to the `CloseFile`, `CheckReadAsync`, `Read`, `ReadAsync`, `GetFhLongevity`, `GetFileStatus`, and `SetFhLongevity` operations. *See also* peek mode and modify mode.

ready state

The state a process is in when it could be running, but a higher priority process is currently running. Any number of processes can be in the ready state at a time. *See also* running state and waiting state.

real mode

One of the CTOS operational modes. In real mode, application programs can address up to one megabyte of memory. *See also* protected mode.

realtime clock

The realtime clock (RTC) is used by the operating system to provide the current date and time of day and timing of intervals (in units of 100 ms). *See also* programmable interval timer.

record fragment

A contiguous area of memory within a record. A record fragment is specified using an offset from the beginning of the record and a byte count.

record number

Specifies the record position relative to the first record in a file. The record number of the first record in a structured file access method file is 1.

record sequential access method (RSAM)

Provides blocked, spanned, and overlapped input and output. An RSAM file is a sequence of fixed-length or variable-length records. Files can be opened for read, write, or append operations. *See also* blocked, record sequential work area, and spanned.

record sequential work area (RSWA)

A 150 byte memory work area for the exclusive use of the record sequential access method procedures. Any number of RSAM files can be open simultaneously using separate RSWAs. *See also* record sequential access method.

recording file

A file used in recording mode. The file contains a copy of all characters typed at the keyboard while recording mode is active. A recording file can later be used as a submit file to repeat the same sequence of input characters. The use of a recording file and the use of a submit file are mutually exclusive. *See also* recording mode and submit file.

recording mode

When recording mode is active, all characters typed at the keyboard and read in character mode are written to a recording file, in addition to being returned to the client process. *See also* recording file.

reentrant code

Code that can be executed by more than one process at the same time. System-common procedures, for example, must be written in reentrant code.

relative address

See offset.

release documentation

Software Release Announcements (SRAs) and the *CTOS System Software Installation and Configuration Guide*.

request

A kernel primitive that directs a request for a system service from a client process to the service exchange of the system service process. Before the primitive is issued, the data required for the system service is arranged in a request block in the client's memory. The easiest way for the client to access the service is to use the request procedural interface, which automatically builds the request block. *See* request procedural interface.

request-based system service

A system service that serves requests submitted by clients. *See also* system service process.

request block

A block of memory provided by a client that contains a special type of message formatted according to specific conventions and used by all interprocess communications to the operating system. The memory address of the request block is provided by the client during a Request primitive and by the system service during a Respond primitive. A request block is the “element” that the application program (or the operating system) sends to the operating system to request that a particular operation be performed.

request code

A 16 bit value that uniquely identifies a system service. For example, the request code for the Write operation is 36. The request code is used both to route a request to the appropriate system service process and to specify to that process which of the several services it provides is currently being requested.

request code level

Sixteen 4K byte request levels are supported in a system request routing table. Each request level contains up to 4K request definitions. To use the request procedural interface and validity checking structures, a request must be defined by a request code in an even-numbered level.

request procedural interface

A convenient way to access system services, compatible with high-level languages, such as C and Pascal, as well as assembly language. The request procedural interface is a routine within the operating system that is executed when a program calls a request-based operation. The routine builds a request block message and calls the Request primitive, while the calling program is placed in the waiting state at its default response exchange for the system service to respond. *See also* default response exchange, request, respond, and wait.

request routing table

An operating system table that defines the file specification and routing for a request. The table contains information on file specifications, routing rules, destination, exchange, and the request procedural interface. Requests served by dynamically installed system services can be added to the request routing table by using the loadable request file. *See also* loadable request file.

response exchange

The exchange at which the requesting client process waits for the response from a request-based system service. *See also* default response exchange and exchange.

RIH

See raw interrupt handler.

RJE

Remote job entry.

ROD

Return overlay descriptor.

ROM

Read-only memory.

RSAM

See record sequential access method.

RSWA

See record sequential work area.

RTC

See realtime clock.

run file

Program image created by the Linker containing object modules bound together into code and data segments.

running state

The state a process is in when the processor is actually executing its instructions. Only one process at a time can be in the running state. *See also* ready state and waiting state.

S

SA

See segment address.

SAM

See sequential access method.

SAMGen

See SAM generation.

SAM Generation

Procedure for specifying which device-dependent SAM object module(s) will be linked with an application program.

sandbar

Obstruction in physical memory that fragments memory and prevents loading an additional program.

SAR

Screen attribute register.

sb string

A string in which the first byte contains the byte count of the string and the remaining bytes contain the actual string.

SCAT

System-common address table.

screen attribute

Controls the presentation of the entire screen. The standard screen attributes are blank, reverse video (dark characters on a light background), half-bright, number of characters per line, and the presence or absence of character attributes. *See also* character attribute.

SCSI

Small Computer Systems Interface, an American National Standard for the interconnection of computers with peripheral devices such as disk drives, tape drives, and printers.

SCSI bus

The physical cable connecting the computer to all the SCSI devices.

SCSI device

A host computer adapter, peripheral controller, or an intelligent peripheral that can be attached to the SCSI bus.

SCSI manager target mode

SCSI mode in which the SCSI Manager operates as a SCSI *target* itself. The SCSI Manager acts as if it were a peripheral device, and other initiators connected to the SCSI bus can send commands and data to it.

SCSI target ID

The encoded representation of the unique address (0 through 7) assigned to a SCSI device. The address is normally assigned and set in the SCSI device during system installation.

SCSI status

One byte of information sent from a SCSI target to an initiator, upon completion of each SCSI command.

sdType

A 6 byte block of memory in which the first 4 bytes contain the address of the parameter, and the last 2 bytes contain the parameter size.

secondary task

A run file that is loaded by the primary task into a partition containing one or more run files. *See also* primary task.

segment

A contiguous area memory area in the linear address space. Segments are usually classified into one of three types: code, static data, or dynamic data. Each kind of segment can be either shared or nonshared. *See also* code segment and data segment.

segment address

The segment base address in linear memory (real mode) or a selector (protected mode).

segment alias

A (protected mode) copy of a segment descriptor returned by the `CreateAlias` operation. The segment type specified by the descriptor can be changed (from code to data, for example) by calling `SetSegmentAccess`. *See also* segment descriptor.

segment base address

Location at which a segment starts. In real mode, the segment base address is the high-order 16 bits of the 20 bit physical memory address. It is determined by multiplying the segment address by 16. In protected mode, the segment base address is contained in the segment descriptor indexed by the logical memory address selector. *See also* descriptor, logical memory address, physical memory address, segment address, and selector.

segment descriptor

A protected mode structure that is indexed by the logical memory address selector. The descriptor contains information about a segment including the maximum offset that can be used in the logical memory address (limit), the segment base address, and the access rights byte. *See also* access rights byte, logical memory address, offset, segment base address, and selector.

segment element

A section of an object module. Each segment element has a segment name.

segment not-present fault

An interrupt (in protected mode) that occurs when an overlay or a 4K byte page is referenced that is not in memory. *See also* segment not-present fault interrupt handler and virtual code management facility.

segment not-present fault interrupt handler

An interrupt (in protected mode) servicing procedure to which control is passed when a not-present fault occurs. *See also* segment not-present fault.

segmented addressing

The type of addressing used by the operating system in which every address is relative to a segment address.

selector

The high-order 16 bits of the logical memory address (segment address) in protected mode. The selector is an index of a segment descriptor in either a local descriptor table or a global descriptor table. *See also* logical memory address, segment address, and segment descriptor.

semaphore

A variable with two distinct states. The paging service uses a semaphore to protect paging data structures from being used by other processes while they are used by the paging service.

send

A kernel primitive typically used for communication between processes in the same partition (user number). Send accepts any 4 byte field as a parameter. This is usually, but not necessarily, the address of a message.

sequential access method (SAM)

Provides device-independent access to a default set of real devices, such as the screen, printer, files, and keyboard. To transfer data to or from the device, SAM uses a character-oriented sequence of bytes known as a byte stream. *See also* byte stream, byte stream work area, communications byte stream, disk byte stream, Generic Print System byte stream, keyboard byte stream, pre-GPS spooler byte stream, printer byte stream, tape byte stream, video byte stream, and X.25 byte stream.

server

A processor with a master agent. A server can be a server workstation or a shared resource processor GP or CP board. *See also* master agent.

server workstation

A server workstation can serve a cluster configuration. The server workstation provides file management, queue management, and other services to all the cluster workstations. In addition, it supports its own interactive programs. *See also* cluster workstation and cluster configuration.

service exchange

An exchange that is assigned to a request-based system service process when the system service is dynamically installed or at system build. The system service process waits for requests for its services at its service exchange.

service process

See system service process.

session

A connection between two nodes in the network initiated by the Net Agent. *See also* Net Agent and Net Server.

sg

A variable prefix indicating that the variable is a global descriptor table selector. *See also* global descriptor table and selector.

shared resource processor linear address

A 4 byte quantity in which the most significant byte is at the lowest address. A linear address is absolute, not segment-based.

shared resource processor routing type

Used as part of a shared resource processor request definition and describes where to route the request.

short-lived memory

Memory existing only for the duration of current program execution. The memory is deallocated when the program terminates.

sixteen-bit addressing

Using a logical address with a 16-bit offset to address memory. This type of addressing allows access of up to 64K bytes of memory.

sizing

Controlling the maximum amount of memory the program can allocate and the minimum amount of memory the operating system will allocate for the program before attempting to run it (protected mode operating systems only). A program can be sized at link time. *See also* variable partition.

sl

A variable prefix indicating that the variable is a local descriptor table selector. *See also* sg, local descriptor table, and selector.

slot number

Refers to a position within a shared resource processor enclosure where a processor can be located. (For an illustration of slot numbering, see “SRP Slot Numbers” in the *CTOS System Administration Guide*.)

SMD

Storage module device.

sn

A variable prefix indicating that the variable is a selector. *See also* selector.

software-loadable cursor

The type of cursor displayed by a bit-map workstation. Unlike the standard cursor on character-map workstations, the software-loadable cursor is changeable by software. *See also* bit-map workstation.

software virtual map

The character map for a bit-map workstation. It is not controlled by video controller hardware. *See also* bit-map workstation.

SP

Stack pointer or storage processor.

spanned

A record file in which a record can begin and end in different physical sectors. *See also* blocked and record sequential access method.

spooled printing

Transfers text to a disk file for temporary storage and queues a request for the spooler to transfer the text to the first available printer interface under control of the spooler. This facilitates sharing of printers by cluster workstations, as well as concurrent, interactive computing and printing.

spooler

A dynamically installed system service that transfers text from disk files to the printer interfaces of the workstation on which the spooler is installed. It can simultaneously control the operation of several printers. A disk-based, priority-ordered queue controlled by the Queue Manager contains the file specifications of the files to be printed and the parameters (such as the number of copies and whether to delete the file after printing) controlling the printing. This allows the spooler to resume printing automatically when reinstalled following an operating system reload. *See also* direct printing, printer byte stream, and pre-GPS spooler byte stream. (For details on the Queue Manager and Pre-GPS spooled printing, see the *CTOS Programming Guide, Volume II*.)

spooler escape sequence

Special character sequences embedded in text files that cause the printer to pause when processed by the spooler. Escape sequences are available to request a forms change, a print wheel change, and a generic printer pause.

sr

A variable prefix indicating that the variable is a paragraph number. SR refers specifically to the real mode form of a segment address. *See also* paragraph, selector, and segment address.

SS

Stack segment.

STAM

Standard access method.

standard character set

The single byte character codes. (See Appendix B in the *CTOS Procedural Interface Reference Manual*.)

standard cursor

The type of cursor on a character-map workstation. It consists of a blinking underline and is not changeable by software.

static data segment

See data segment.

status code

Reports the status of the requested operation (for example, its success or failure). A status code is stored in a request block by the system service process and is examined by the client process.

stealable

Describes an available cache entry on the LRU list.

sticky

Describes a cache entry that is to remain in the cache (is not subject to removal from the cache according to the LRU algorithm).

structured file access methods

Object module procedures that are linked to application programs as required to augment the capabilities of the file management system. They provide buffering and use the asynchronous I/O capabilities of the file management system to automatically overlap I/O and computation. *See also* Indexed Sequential Access Method, record sequential access method, and direct access method.

stub

A 5 byte structure used to account for a program procedure in an overlay program. In real mode, the first byte is either a JMP or a CALL instruction; in protected mode, the first byte is always a JMP. The remaining 4 bytes are a procedure address.

submit file

A file containing the same sequence of characters that would be typed to the application reading the keyboard. Playback of the file is activated by a playback request from the application or a command from the Executive. In playback mode, the system input process reads the character from the submit file rather than from the keyboard. *See also* recording file and system input process.

submit file escape sequence

Consists of two or three characters. The first is the code 03h, which indicates the presence of an escape sequence. The second character of the escape sequence is a code to identify the special function. The third character, if present, is an argument to the function. *See also* escape sequence and submit file.

submit mode

One of three modes of operation used by the system input process. In submit mode, input is read from the submit (recorded) file rather than from the keyboard.

subparameter

A user-supplied string in a parameter field of an Executive command form. A subparameter is stored in the variable length parameter block so it is available for use by other programs that will execute in the same partition as the Executive. *See also* variable length parameter block.

swap

To copy a partition (user number) into memory or out of memory to a disk file. Swapping is managed by a partition managing program on multipartition operating systems or by the operating system itself on variable partition systems.

swapping request

A system request issued to system service(s) whenever the operating system is going to suspend an application and swap it to disk. Swapping requests ensure that no responses are made to clients in an application that is not resident in memory.

synchronous mode

See synchronous operation.

synchronous operation

A procedure, or protocol, that requires a response at an exact time. The Wait primitive, for example, can be issued by a program to wait at an exchange so it can synchronize its operation with the system service's response.

Sys directory

The Sys directory of each volume contains entries for system files, including the bad sector file, the file header blocks, the master file directory, the system image, the crash dump area, the log file, and the Executive. The Sys directory is created by the **Format Disk** command (described in the *CTOS Executive Reference Manual*) rather than by the CreateDir operation or the **Create Directory** command. *See also* Sys volume.

Sys volume

The volume from which the operating system is bootstrapped. The Sys directory of the Sys volume contains entries for system files that are not necessary in the Sys directories of other volumes. These additional entries must be placed in [Sys]<Sys> when the volume is initialized. *SysImage.sys*, *CrashDump.sys*, and *Log.sys* are created (but not initialized) by the **Format Disk** command (described in the *CTOS Executive Reference Manual*). The other file entries are created using the CreateDir operation or the **Create Directory** command. These system files are the system images, the crash dump areas, the log file, the Debugger, the Executive, the Executive command file, and the standard character font. *See also* crash dump area, Executive, log file, Sys directory, and system image.

Sys.cmds

The Executive command file containing information about each command known to the Executive. [Sys]<Sys>*Sys.cmds* is used if there is no *Sys.cmds* file in the application system control block. The **New Command** command or the **Command File Editor** (which are described in the *CTOS Executive Reference Manual*) are used to enter additional commands into *Sys.cmds*.

Sys.keys

A file containing a keyboard mapping table that maps keyboard codes to character codes on BTOS operating systems only.

SysGen

See system generation.

system build

The collective name for the sequence of actions necessary to construct a customized system image. System build allows the specification of installation-specific parameters and the inclusion of user-written system services. Note that system services usually are installed dynamically (that is, without regenerating the system) if they are not part of the operating system itself. *See also* dynamically installed system service.

system-common NLS table area

The area in main memory where the NLS tables are loaded at boot time and made available to programs. *See also* NLS table.

system-common procedure

A system-common procedure performs a common system function, such as returning the current date and time. The code of the system-common procedure is included in the system image and is executed in the same context and at the same priority as the invoking process.

system-common service

A system service process that contains system-common procedures. *See also* System service process.

system configuration block

A system structure allowing the application to determine detailed information about the system image (workstation configuration and system build parameters).

system event

An event affecting the executability of a process. Examples of system events are an interrupt from a device controller, multibus device, timer, or realtime clock, or a message sent from another process. The system event causes a message to be sent to an exchange at which a higher priority process is waiting; this, in turn, causes the operating system to reallocate the processor. *See also* event.

system generation

System generation (also called SysGen) is the process of building a customized version of the operating system by changing the default values of parameters or by removing operating system functionality. (For details, see your release documentation and the *CTOS System Administration Guide*.)

system image

Contains a run file copy of the operating system
(/Sys/<Sys>SysImage.sys).

system input process

Permits a sequence of characters from a file to be substituted for characters typed at the keyboard. The use of submit files allows the convenient repetition of command sequences. *See also* submit file.

system partition

Contains the operating system or dynamically installed system services. *See also* application partition.

system profile

The keyboard emulated on an operating system. (Applications running on the operating system assume this keyboard is the one being used.)

system request

An abort, change user number, swapping, or termination request.

system service

An operation performed by a system service process.

system service process

An operating system process that provides services to client processes. System service processes are of two types: request-based system services and system common services. Request-based system service processes serve requests submitted by client processes throughout the network; whereas, system-common services contain system-common procedures that can be used by clients at the local workstation. Both Unisys- and user-written system service processes can be dynamically installed or linked with the system image at system build.

T

task

See primary task and secondary task.

task state segment (TSS)

A segment (in protected mode) that holds the contents of registers for a process while the processor is executing a different process.

task switch

A context switch on a protected mode operating system. *See* context switch.

terminal output buffer

A system structure used by a shared resource processor to manage I/O to 8251 ports.

termination request

A system request issued to notify system services of a terminating client. Upon notification, the system services can release resources, such as open files and locked ISAM records, allocated to the terminating client.

text file

A file in which each byte represents a printable character or a control character such as Tab (09h), New Line (0Ah), or Formfeed (0Ch).

thirty-two bit addressing

Using a logical address with a 32-bit offset to address memory. This type of addressing allows access of up to 4 gigabytes of memory.

thrashing

Frequent disk activity as a result of the working sets exceeding the size of physical memory. Local thrashing results when a program is greater than the partition size or all of physical memory.

time slicing

When processes with the same priority are executed in turn for intervals of 100 ms in round-robin fashion. Processes having priorities within a predefined range are subject to time slicing.

timer request block (TRB)

A data structure shared by the client process and timer management. The TRB defines the interval after which a message is to be sent to a specified exchange. *See also* realtime clock.

toggle

A characteristic of some chord keys (for example **LOCK**) where the key maintains an influence on other keys after it has been pressed and released. To return to its original state (no influence on other keys), the toggle key must be pressed a second time. By default, chords that do not toggle (for example **SHIFT**) maintain their influence only while pressed.

TP

Terminal processor.

TPIB

Timer pseudointerrupt block.

translating

Generating the character code from an unencoded value and the chord state.

transport protocol layer

A layer of the SCSI standard that defines how logical connections are established between SCSI devices; how commands, data, and status are transferred between devices; and how the integrity of this information is assured.

trap

See internal interrupt.

trap gate

An entry point (in protected mode) in the interrupt descriptor table of a trap handler. Trap gates on 80386-based and higher systems support virtual 8086 mode. *See also* interrupt descriptor table and trap handler.

trap handler

An interrupt procedure that handles internal interrupts. *See* internal interrupt.

TRB

See timer request block.

TRUE

Represented in a flag variable as 0FFh.

trap

An internal interrupt.

type-ahead buffer

Stores unencoded keyboard values to be returned directly or after translation to an application. The type-ahead buffer also stores other input event types such as mouse events.

type checking

A protection feature supported in protected mode that controls the type of memory a program can access.

U

UCB

See user control block.

unencoded mode

Reading keyboard values before translation to a character code. The application receives an indication of each key depression and release. *See also* mapped unencoded mode, raw unencoded mode, and character mode.

user control block (UCB)

A memory-resident system structure for a user number. The UCB contains the default volume, default directory, default password, and default file prefix set by the last SetPath and SetPrefix operations.

user file block

A structure containing a pointer to the file control block for each open file.

user number

A 16 bit identifier that uniquely identifies the program(s) and/or the resources (such as file handles, short-lived memory, long-lived memory, and exchanges) associated with a partition. User numbers are historically known as partition handles.

user structure

A partition component that contains a loose collection of information describing an application partition. It can contain the following partition structures: the partition configuration block, extended partition descriptor, application system control block, and optionally the batch control block.

V**VAM**

See Video Access Method.

variable length parameter block (VLPB)

A partition structure used by the Executive to communicate parameters to a succeeding application in the partition in which the VLPB is located. The VLPB is created in the long-lived memory of an application partition, and its memory address is stored in the application system control block. *See also* application system control block.

variable partition

A memory partition that can use up to the maximum amount of memory specified at link time (when the program to be loaded into the partition was sized). *See also* fixed partition and sizing.

VCB

See video control block.

VDM

See video display management.

verify code

The number of times that the server has been rebooted.

VHB

See volume home block.

Video Access Method (VAM)

An installable system-common service providing direct access to the characters and attributes of each video frame. VAM can put a string of characters anywhere in a frame, specify character attributes for a string of characters, scroll a frame up or down a specified number of lines, position a cursor in a frame, and reset a frame.

video attributes

Control the visual presentation of characters on the screen. There are two kinds of video attributes: screen and character. *See also* character attribute and screen attribute.

video byte stream

A byte stream that uses the video display. *See also* byte stream and byte stream work area.

video capability

Screen characteristics provided by character-map and bit-map workstations. These characteristics can be obtained programmatically based on character cell size for a workstation by calling the QueryVidHdw or the QueryVideo operation. *See also* bit-map workstation, character cell, and character-map workstation.

video control block (VCB)

A system structure containing all information known about the video display, including the location, height, and width of each frame, and the coordinates at which the next character is to be stored in the frame by the sequential access method. The address of the VCB can be obtained by calling the GetPStructure operation. *See also* frame descriptor.

video display management

A set of video operations that provide direct control over the way that the video appears. With these operations, an application can determine the level of video capability, load a new character font into the font RAM, change screen attributes, stop video refresh, calculate the amount of memory needed for the character map based on the desired number of columns and lines and the presence or absence of character attributes, initialize each of the frames, and initialize the character map.

video refresh

A hardware function that reads characters and character attributes from the character map in memory. It then converts them from the extended ASCII (8 bit) memory representation to a bit array by accessing the font RAM and displays these bits on the screen as a pattern of dots (pixels). *See also* font RAM.

Virtual Code Management facility

The method of demand segmentation of code segments supported on operating systems prior to virtual memory operating systems. The virtual code management facility divides the code of an overlay program into variable-length segments that reside on disk in a run file. As the overlay program executes, only those code segments that are required at a particular time actually reside in the main memory of the application partition; the other code segments remain on disk until they, in turn, are required. When a code segment is no longer required, it is discarded. The discarded overlay is not read back to disk. *See also* code segment and virtual memory.

virtual 8086 mode

An Intel mechanism on 386-based and higher microprocessors that allocates memory regions and assigns each the operating system characteristics of an 8086 microprocessor. Each memory region, thus, provides one-megabyte of addressability to the application executing in it. (For additional information, see the Intel manuals.)

virtual memory

The apparent size of memory (to an application), which is greater than the physical memory size. Paging and segment swapping are two memory management techniques that create virtual memory. (Using program overlays does not create virtual memory because overlays are not transparent to the application.) *See also* virtual code management facility.

VLPB

See variable length parameter block.

volume

The medium of a disk drive that was formatted and initialized with a volume name, a password, and volume control structures such as the volume home block, the file header blocks, the master file directory, and so forth. A floppy disk and the medium sealed inside a hard disk are examples of volumes.

volume control structures

Structures allowing the file management system to manage (allocate, deallocate, locate, avoid duplication of) the space on the volume not already allocated to the volume control structures themselves. A volume contains a number of volume control structures: the volume home block, the file header blocks, the master file directory, and the allocation bit map, among others.

volume home block (VHB)

The disk-resident root structure (that is, the starting point for the tree structure) of information on a disk volume (or in a disk partition). The VHB contains information about the volume such as its name and the date it was created. The VHB also contains the memory addresses of the log file, the system image, the crash dump area, the allocation bit map, the master file directory, and the file header blocks.

volume password

Protects a volume.

W**W-block**

An area of memory used to hold very large inter-CPU communications messages. *See also* Y-block and Z-block.

wait

The kernel primitive that a client calls to be placed in the waiting state. *See* waiting state.

waiting state

The state a process is in when it is waiting at an exchange for a message. A process enters the waiting state when it must synchronize with other processes. A process can only enter the waiting state by voluntarily issuing a Wait kernel primitive that specifies an exchange at which no messages are currently queued. The process remains in the waiting state until another process (or interrupt handler) issues a Send (or PSend, Request, or Respond) kernel primitive that specifies the same exchange that was specified by the Wait primitive. Any number of processes can be in the waiting state at a time. *See also* ready state and running state.

working set

Number of pages a program needs to make reasonable forward progress.

workstation agent

Code that converts interprocess requests to interstation messages for transmission to the server. The workstation agent is included at system build in a system image that is to be used on a cluster workstation. The workstation agent code responds to master agent polling by sending a request to the server or by informing the server that it has no request to send. *See also* server and master agent.

write-back

Describes a cache that can contain data not matching what is in secondary storage.

write-behind

A mode in which the direct access method writes changed sectors of the buffer to disk only when new sectors are brought into the buffer, the direct access method file is closed, or the mode is changed to write-through. Write-behind mode provides better performance when the direct access method is used to modify records in sequential order.

write-through

A mode in which secondary storage is kept in synchronization with the data in memory. A write-through cache immediately writes the data back to disk at the completion of each request made to modify data. In write-through mode, the direct access method immediately writes the changed sectors of the buffer to disk whenever a record is written or deleted. This guarantees that the file content on disk is accurate at the completion of a modify operation.

X

X-block

An area of memory allocated for cluster communications.

X-Bus interrupt handler

A routine that services interrupts generated by modules connected to the X-Bus. Three levels of interrupt handlers are provided: XINT0, XINT1, and XINT4. XINT0 and XINT1 are provided for faster servicing and are nonshareable among modules. XINT4 is sharable and, thus, is slower.

X-Bus memory master

A device that controls the X-bus for a given activity, such as a DMA transfer. The device can access the main processor RAM, but the processor cannot access the module's memory address space.

X-Bus memory master/slave

A device that can access the main processor RAM and in which the processor can access the module's memory address space.

X-Bus memory slave

A device that cannot access the main processor RAM, but the processor can access the module's memory address space.

X.25 byte stream

A byte stream that enables data transmission by means of the X.25 Network Gateway. Each open X.25 byte stream corresponds to a virtual circuit that is initiated when the byte stream is opened and cleared when the byte stream is closed. *See also* byte stream and byte stream work area.

Y

Y-block

An area of memory used to hold inter-CPU communications messages longer than 512 bytes. *See also* W-block and Z-block.

Z

Z-block

An area of memory used to hold inter-CPU communications messages of less than 512 bytes. *See also* Y-block and W-block.

Index

- !Sys, 12-28
- \$ Directory, 12-42
- +Sys, 12-29
- .def file (*See* module definition file)
- 16-bit addressing, 24-1
- 32-bit addressing, 24-1
- 80286 task state segment extension, 27-1
- 80386 task state segment extension, 27-1
- :fSuppressGlobalPolicy:, 3-14

A

- abort requests, 33-18
- accessing
 - and modifying system date and time, 26-22
 - commands, 26-18
 - communication ports, 15-2
 - disk devices, 13-1
 - files, 12-4, 12-8, 12-28, 20-2, 20-4
 - global data, 39-12
 - memory of another processor, 42-5
- NLS tables, 45-17
- resources in disk files, xli, 26-2 to 26-14
- resources in memory, xlii, 26-14
- system services, 4-7, 30-30, 30-31 to 30-32
- volume home blocks, 13-4
- X-Bus module memory, 41-3

- X-Bus modules in protected mode, 41-5
- X-Bus modules in real mode, 41-5
- accessing system services
 - using the kernel primitives, 4-7, 30-31 to 30-32
 - using the request procedural interface, 30-30
- accessing NLS tables
 - alternate tables linked with application, 45-17
 - tables in application memory, 45-17
 - tables loaded at system initialization, 45-17
- AcquireByteStreamC, 15-5, 15-6
- Action, 10-34, 11-12, 11-45, 11-52 to 11-53
- adding passwords to files, 12-13
- adding resources to files, 26-11
- address mapping, 39-2
- addressability, 36-11
- addressing
 - 16-bit, 24-4
 - 32-bit, 24-4
 - a byte in a segment, 24-3
 - memory, 4-10
 - protected mode, 24-3
 - real mode, 24-3
 - segment base addresses, 24-3
- addressing memory, 4-10
- addressing, 16-bit, 24-4
- addressing, 32-bit, 24-4

- advantages of dynamic link
 - libraries, 39-21
- advantages of using
 - device-dependent interfaces, 8-2
 - sequential access method, 8-1
- aliasing, 34-5
- AllocAllMemory, 24-13
- AllocAllMemorySL, 24-9
- AllocAreaSL, 24-9, 24-13
- AllocateSegment, 2-13, 24-11, 24-17
- allocating
 - response exchanges, 30-31
 - global linear address space, xl
 - memory, 2-22
- AllocCommDmaBuffer, 42-6, 42-8
- AllocExch, 30-54, 33-9
- AllocHugeMemory, 2-13, 24-11, 24-12, 24-17
- AllocMemoryFramesSL, 24-13
- AllocMemoryLL, 24-9, 24-14, 6-8
- AllocMemorySL, 24-9, 24-13, 33-9
- allowed states, 11-2, 11-23
- allowed states table, 27-1
- alternate request procedural
 - interface, 30-30
- application profile keyboard, 11-2, 11-43
- application system control block, 6-3, 27-1, 36-5
- ASCB (*See* application system control block)
- AsGetVolume, 35-14
- Assembler, 4-1
- AsSetVolume, 35-14
- assigning
 - I/O addresses to X-Bus modules, 41-1
 - process priorities, 2-2
 - volume passwords, 12-12
- AssignKbd, 36-22
- AssignVidOwner, 36-22
- asynchronous operation, 20-1
- asynchronous RS-232-C
 - communications, 15-4
- Asynchronous System Service
 - operations
 - AllocMemoryInit, 35-2
 - AsyncRequest, 35-2
 - AsyncRequestDirect, 35-2
 - BuildAsyncRequest, 35-2
 - BuildAsyncRequestDirect, 35-2
 - CheckContextStack, 35-2
 - CreateContext, 35-2
 - HeapAlloc, 35-2
 - HeapFree, 35-2
 - HeapInit, 35-3
 - LogMsgIn, 35-3
 - LogRequest, 35-3
 - LogRespond, 35-3
 - ResumeContext, 35-3
 - SwapContextUser, 35-3
 - TerminateAllOtherContexts, 35-3
 - TerminateContext, 35-3
 - TerminateContextUser, 35-3
- AtFileInit, 12-44
- AtFileNext, 12-44
- atomic, 31-3, 31-5
- Audio Service operations
 - AsSetVolume, 35-14

automatic data segment, 39-13
automatic target mode functions,
18-21
automatic volume recognition,
12-11
automatic pause between full text
frames, 10-5, 10-7
AVR (*See* automatic volume
recognition)

B

backing store, 3-2
Batch control block, 27-2, 36-5
Beep, 11-56
binary mode, 8-7
binary resource file, 26-3
binding (*See* linking)
binding system-common procedures,
4-4
bit-map workstations, 10-1
blocked records, 20-1
blocks
 storing copies of request blocks,
 32-2
 transferring ICC messages, 32-5
boot block, 27-2
bootable volumes, 43-1
bsKbd, 8-3, 8-4
bsVid, 8-3, 8-4
BSWA (*See* byte stream work area)
buffer ownership table, 32-6, 32-7
buffers, 25-3
 DAM, 23-2
 caller-supplied byte stream, 8-3
 ownership table, 32-6, 32-7
 management modes, 23-3

BuildFileSpec, 12-37 to 12-38,
12-48
BuildFullSpecFromPartial, 12-48
building
 and parsing file specifications,
 12-35 to 12-38
 customized operating systems,
 43-1
 request blocks, 33-4
 single-line text editors, 26-27
 system keyboard files, 11-32
building and parsing file
 specifications, 12-35 to 12-38
BuildSpecFromDir, 12-48
BuildSpecFromFile, 12-48
BuildSpecFromNode, 12-48
BuildSpecFromPassword, 12-48
BuildSpecFromVol, 12-48
built-in networking, 1-2, 1-4
bus address, 42-1
bus addresses, 32-2, 32-6
 on shared resource processors,
 42-3 to 42-5
 users, 42-5
byte stream work area, 8-1
byte streams, 8-3, 8-4, 9-1

C

cache entry descriptor, 25-5, 27-2
cache pool descriptor, 25-4, 27-2
cache pool handle, 25-4
cache statistics block, 25-13
CacheClose, 25-13, 25-14
CacheFlush, 25-12, 25-14
CacheGetEntry, 25-9, 25-14
CacheGetStatistics, 25-13, 25-14
CacheGetStatus, 25-12, 25-14

- CacheInit, 25-14
- CacheReleaseEntry, 25-9, 25-14
- caching, 1-4, 1-5
- calculating cache entries, 25-6
- calculating request sizes, 32-9
- call gates, 34-7
- calling DLL procedures, 39-11
- CdAbsoluteRead, 35-5
- CdAudioCtl, 35-5
- CdClose, 35-5
- CdControl, 35-4
- CdDirectoryList, 35-4
- CdGetDirEntry, 35-4
- CdGetVolumeInfo, 35-4
- CdOpen, 35-5
- CdRead, 35-5
- CdSearchClose, 35-4
- CdSearchFirst, 35-4
- CdSearchNext, 35-4
- CdServiceControl, 35-5
- CDT (See CPU description table)
- CdVerifyPath, 35-4
- CdVersion, 26-38
- CD-ROM Service operations
 - CdAbsoluteRead, 35-5
 - CdAudioCtl, 35-5
 - CdClose, 35-5
 - CdControl, 35-4
 - CdDirectoryList, 35-4
 - CdGetDirEntry, 35-4
 - CdGetVolumeInfo, 35-4
 - CdOpen, 35-5
 - CdRead, 35-5
 - CdSearchClose, 35-4
 - CdSearchFirst, 35-4
 - CdSearchNext, 35-4
 - CdServiceControl, 35-5
 - CdVerifyPath, 35-4
- CfaFFVersion, 26-38
- CfaServerVersion, 26-38
- CfaWaVersion, 26-38
- Chain, 5-8, 33-11
- change user number requests,
 - 33-20
- ChangeCommLineBaudRate, 16-3,
 - 16-5
- ChangeFileLength, 12-7, 12-22,
 - 12-44
- ChangeOpenMode, 12-46
- ChangePriority, 33-9
- changing
 - file passwords, 12-13
 - and querying SCSI path parameters, 18-9
 - keyboards, 11-42
 - or removing a directory password, 12-12
 - processor modes, 1-6
- character
 - attributes, 10-2
 - cell sizes, 10-14
 - code, 11-2
 - extended set, 11-3
 - mode, 8-8, 11-2
 - multibyte, 11-4
 - plus, 11-12, 11-13
 - repeating, 45-6
 - standard, 11-4
- character attributes, 10-2
- character code, 11-2
- character mode, 8-8, 11-2
- character plus, 11-12, 11-13
- character repeating, 45-6
- character set
 - extended, 11-3
 - standard, 11-4

- character-map workstations, 10-1
- Check, 30-17, 30-54
- CheckErc, 5-6, 5-7
- CheckForOperatorRestartC, 15-5, 15-7
- checking SCSI operation error
 - status, 18-16 to 18-18
- CheckpointBs, 8-16
- CheckpointBsAsyncC, 15-8
- CheckPointBsC, 15-7
- CheckpointRsFile, 22-3
- CheckpointSysIn, 11-57
- CheckReadAsync, 12-49, 30-30
- checksum, 29-4
- CheckWriteAsync, 12-49, 30-30
- chord information structure, 27-2
- chord state, 11-2
- chords, 11-2, 11-23
- chords supporting table, 11-30 to 11-31
- cleaning pages, 3-2, 3-11
- cleaning up resources, 39-15
- clearing semaphores, 31-3, 31-8
- ClearPath, 12-45
- clients, 30-1, 30-2, 30-53, 33-1, 34-4, 39-1
- clock algorithm, 3-9 to 3-11
- CloseAllFiles, 12-22, 12-43
- CloseAllFilesLL, 12-22, 12-46
- CloseAltMsgFile, 45-34
- CloseByteStream, 8-15
- CloseDaFile, 23-4
- CloseErcFile, 26-20, 26-21, 26-35
- CloseFile, 12-22, 12-43
- CloseMsgFile, 45-33
- CloseRsFile, 22-3
- CloseRtClock, 37-3, 37-8
- CloseServerMsgFile, 45-26, 45-35
- CloseSysCmds, 26-18, 26-34
- CloseVidFilter, 8-4, 9-5
- closing
 - cache pools, 25-13
 - command files, 26-18
 - error message files, 26-21
 - files, 12-26
- closing cache pools, 25-13
- closing command files, 26-18
- closing error message files, 26-21
- closing files, 12-26
- CLRB (*See* communications line return block)
- cluster, 2-6
 - communications channels, 44-1
 - configuration, 2-6, 30-32, 30-33, 44-1
 - processor, 2-7
 - workstations, 2-6, 2-7
- cluster communications channels, 44-1
- cluster processor, 2-7
- CMIH (*See* communications mediated interrupt handlers)
- code segment, 24-4, 24-6
- code sharing, 1-7, 29-1
- COFF section descriptor, 27-2
- collapsing boxes, 26-16
- color programming, 10-17
- Comm Nub, 40-16
- Command Access Service operations
 - ObtainAccessInfo, 35-6
 - ObtainUserAccessInfo, 35-6
- command
 - bit, 11-16
 - case, 6-5
 - form, 6-2
 - interpreter (*See* Executive)
 - masks table, 27-2

- command interpreter (*See* Executive)
- command masks table, 27-2
- commands
 - obtaining information about, 26-18
 - opening a command file, 26-18
- common object file format, 27-2
- communication between application partitions, 36-17
- communications
 - byte streams, 8-8
 - channels, 2-6, 5-6, 15-3, 16-2
 - channel identifiers, 8-11
 - configuration descriptor, 27-2
 - line return block, 16-2
 - parameters, 15-5
 - port expander specifications, 8-12
 - programming, 15-1 to 15-5
 - raw interrupt handlers, 40-16
 - status buffer, 27-2
- communications line return block, 16-2
- communications programming, 15-1 to 15-5
- communications mediated interrupt handlers, 40-18, 40-19
- communications raw interrupt handlers, 40-16
- communications status buffer, 27-2
- CompactDateTime, 26-22, 26-36
- comparing
 - DLLs to other CTOS services, 39-3
 - logical addresses, 26-28
 - strings, 26-14 to 26-15
- compatibility, 45-6
- concurrent file access, 2-5
- conditions table, 27-3
- conditions, 11-23
- ConfigCloseFile, 26-25, 26-40
- ConfigGetNextToken, 26-25 to 26-26, 26-40
- ConfigGetRestOfLine, 26-25, 26-40
- ConfigOpenFile, 26-25, 26-40
- ConfigQueryFilePosition, 26-25, 26-26
- ConfigQueryPosition, 26-40
- ConfigSetFilePosition, 26-25, 26-26
- ConfigSetPosition, 26-40
- configuration files, 8-10 to 8-10
- configuring
 - line speeds, 44-1
 - system paging parameters, 3-14
 - systems to use DLLs, 39-16
- context switch, 29-3, 29-4, 34-5
- contiguous memory regions,
 - managing 24-11
- control chords table, 27-3
- control register, 16-2
- ControlInterrupt, 40-30
- controlling
 - character attributes, 10-5, 10-6
 - character repeat rate, 11-45
 - external interrupt occurrences, 40-8
 - frame allocation, 3-14
 - screen attributes, 10-5
 - scrolling and cursor positioning, 10-5, 10-6
 - semaphore wait semantics, 31-13
- controlling character attributes, 10-5, 10-6
- converting between date and time formats, 26-22
- converting NLS keyboard tables, 11-32
- ConvertToSys, 33-10, 33-28, 34-8, 36-9

- copy data file resources, 26-11, 26-13
- CopyFile, 26-28, 26-43
- copying files, 26-28
- copying multiple resources, 26-12
- CP (*See* cluster processor)
- CP boards, 17-1
- CParams, 6-4, 6-10
- CPU description table, 27-3, 32-2, 32-6
- Crash, 5-7
- Create Keyboard command, 11-16
- Create Keyboard Data Block
 - command, 11-26
- CreateAlias, 24-15
- CreateBigPartition, 36-8, 36-12, 36-15, 36-22
- CreateDir, 12-7, 12-45
- CreateExecScreen, 26-43
- CreateFile, 12-7, 12-9, 12-23, 12-43
- CreatePartition, 36-8, 36-12, 36-22
- CreateProcess, 33-9
- CreateUser, 36-22
- creating
 - bit-map fonts, 10-1
 - bootable volumes, 43-1
 - character-map fonts, 10-1
 - loadable request files, 33-16 to 33-17
 - module definition files, 39-10, 39-14
 - SCSI paths, 18-6
- creating a SCSI path
 - changing and querying path parameters, 18-9
 - default timeout, 18-8
 - defining unique paths, 18-6, 18-7
 - disconnection permission, 18-8
 - read-only SCSI path parameters, 18-9
 - specifying path parameters, 18-8
 - using the path handle, 18-9
- creating and accessing files
 - using byte streams, 12-20, 12-24, 12-26
 - using file management operations, 12-20, 12-22, 12-24, 12-25 to 12-26
 - using structured file access methods, 12-20
- creating bit-map fonts, 10-1
- creating character-map fonts, 10-1
- creating loadable request files, 33-16 to 33-17
- creating module definition files, 39-10, 39-14
- creating partitions, 36-8, 36-15 to 36-16
- CRIH (*See* communications raw interrupt handlers)
- critical section semaphores, 31-3, 31-5, 31-8
 - as an alternative to disabling interrupts, 31-8
 - compared to noncritical, 31-9
 - effect on suspending processes, 31-10
- critical section, 31-5
- CSubParams, 6-4, 6-10
- CTOS foundation
 - event-driven, priority-ordered process scheduling, 1-2
 - messaged-based operation, 1-2
 - multiprogramming, 1-2
 - multitasking, 1-2
 - nationalization, 1-2
 - networking, 1-2

- CTOS I, xliii, 1-1
- CTOS II, xliii, 1-1
- CTOS III, xliii, 1-1, 1-7
- CTOS III enhancements
 - demand paging, 1-7
 - dynamic link libraries, 1-7
 - name management, 1-8
 - semaphores, 1-8
- CTOS, 1-2
- Ctos.lib, xxxviii
- CTOS/XE, 1-2 to 1-2
- CtosToolkit.lib, xxxviii
- CurrentOsVersion, 4-15, 27-10
- customizing
 - keyboard data blocks, 10-37, 11-31 to 11-36
 - SAM object modules, 8-3
 - system services, 2-4
 - user interfaces, 26-16 to 26-16
- customizing data blocks
 - application-specific blocks, 11-32
 - by converting NLS keyboard tables, 11-32

D

- DAM (*See* direct access method)
- DAM buffer, 23-2
- data control structure, 27-3
- data files, 26-3, 26-9
- data processor, 2-7
- data register, 16-2
- data segment selector
 - expand down, 24-5
 - expand up, 24-5
- data, sharing, 29-1
- DAWA (*See* direct access work area)

- DCB (*See* device control block)
- DCXVersion, 26-38
- DDS (*See* Digital data storage)
- DeallocAliasForServer, 24-16
- DeallocateRods, 38-17
- DeallocateSegment, 24-11, 24-17
- deallocating
 - DMA buffers, 42-7
 - system resources, 5-6
 - user numbers, 36-8
- DeallocExch, 30-55
- DeallocHugeMemory, 24-11, 24-12, 24-17
- DeallocMemoryLL, 24-9, 24-14
- DeallocMemorySL, 24-9, 24-13
- DeallocRunFile, 36-18, 36-23
- DeallocSg, 24-15
- decoded value, 11-2
- decoding offsets table, 27-3
- decoding table, 27-3
- default devices, 8-2
- default response exchange, 30-2, 30-18
- DefineInterLevelStack, 24-14
- DefineLocalPageMap, 24-14
- defining
 - diacritic key pairs, 11-40
 - DLL segments, 39-17
 - SRP request routing, 32-3 to 32-5
 - system service requests, 30-27
 - toggle keys, 11-40
 - unique SCSI paths, 18-6, 18-7
- Delay, 37-2, 37-8
- delaying address mapping, 39-9
- DeleteByteStream, 9-5
- DeleteDaRecord, 23-4
- DeleteDir, 12-45
- DeleteFile, 12-43

- demand paging, xl, 1-7, 3-4, 2-20, 42-3
- determining
 - error message length, 26-20
 - monitor resolution, 26-28
 - existence of I-Bus devices, 41-2
- development utilities
 - Assembler, 4-1
 - Librarian, 4-1, 34-11
 - Linker, 2-13, 4-1
- device
 - access to disk files, 13-1
 - control blocks, 12-35, 27-3
 - handler processes, 40-7
 - handlers, 2-5, 40-5
 - interrupt handlers, 40-7
 - interrupts, 40-3
 - names, 13-2
 - passwords, 12-8, 12-13, 13-2
 - specifications, 13-2
- device control block, 12-35, 27-3
- device/file specifications, 8-10 to 8-13
- DeviceInService, 40-30
- devices
 - default, 8-2
 - masking, 40-10
 - patience, 40-9
- device-dependent
 - access to the video, 10-3
 - interfaces, 8-2
 - programs, 7-3
- device-independent
 - access to the video, 10-3
 - programs, 7-3
- device-specific operations, 9-2
 - SetImageMode, 9-2
 - GetBsLfa, 9-2
 - PutBackByte, 9-2
 - QueryVidBs, 9-2
 - SetBsLfa, 9-2
- diacritic keys, 11-15
- diacritic masks table, 27-3
- diacritic, 11-3, 11-15
- diacritical key handling, 45-7
- diacritics table, 27-3
- digital data storage, 8-9
- digital system process, 27-4
- direct access method, 12-3, 20-4, 23-1
- direct access work area, 23-2
- directing data to a byte stream, 26-16 to 26-17
- directory
 - creating, 12-7
 - defined, 12-7
 - maximum files, 12-7
 - names, 12-7
 - passwords, 12-8, 12-12
 - protection, 12-7
 - specifications, 12-9
- directory passwords, 12-8, 12-12
- directory specifications, 12-9
- dirty pages, 3-2
- DisableActionFinish, 11-57
- DisableCluster, 44-4
- disabling interrupts, 31-8
- DiscardInputBsC, 15-6
- DiscardOutputBsC, 15-6
- disk
 - byte streams, 8-5
 - devices, 13-2
 - partitions, 13-4
- DismountVolume, 13-5
- dispatching
 - communications interrupt handlers, 40-16
 - dynamic link library cleanup procedures, 39-15

- distributed environment, 2-6
- DLL (*See* dynamic link library)
- DMA buffer mapping, 42-3
- DmaMapBuffer, 42-6, 42-7, 42-9
- DmaUnmapBuffer, 42-9
- doorbell interrupt, 32-2, 32-7, 32-9, 32-11, 32-12
- Doze, 37-2, 37-8
- DP (*See* data processor)
- DSP control structure, 27-4
- duplication of volume control
 - structures, 2-5
- dynamic binding (*See* dynamic linking)
- dynamic data segment, 24-5
- dynamic link libraries, xlii, 1-7, 39-1
 - advantages, 39-21
 - and the Linker, 39-5
 - cleaning procedures, 39-15
 - compared to request-based services, 39-6
 - compared to system-common services, 39-4 to 39-6
 - execution model, 39-4
 - guidelines for writing, 39-12 to 39-15
 - initializing procedures, 39-14
 - using resources, 39-11
- dynamic linking, 2-24
 - at runtime, 39-20 to 39-21
 - how it works, 39-6 to 39-8
 - impact on loading, 39-8 to 39-9
 - loadtime, 39-1
 - runtime, 39-1
- dynamically customizing keyboard
 - data blocks, 11-33

- dynamically installable
 - services, 2-3
 - system-common procedures, 34-1
- dynamically redirecting video byte streams, 10-5, 10-6

E

- EAR, 41-5 (*See* extended address register)
- economizing on
 - memory usage, 39-4
 - processor time, 31-5
- editing a submit file, 11-50
- emulating LEDs, 11-31
- emulating, 11-3
- emulation
 - defined, 11-28
 - examples, 11-28 to 11-29
 - data block headers, 27-4
 - data blocks in writable segments, 2-23
 - LEDs table, 27-4
- emulation data blocks, 11-26
- emulation LEDs table, 27-4
- emulation table, 27-4
- EnableSwapperOptions, 38-17
- enabling and disabling interrupts, 40-9
- enclosures, SRP, 32-2
- encrypted passwords, 12-18
- encrypting volumes, 12-18
- ending a resource session, 26-8
- end-of-interrupt, 40-10
- ENLS (*See* extended native language support)
- ENLS operations, 45-19 to 45-22

- Enls.lib, xxxviii
- EnlsAppendChar, 45-31
- EnlsCase, 45-20, 45-30
- EnlsCbToCCols, 45-22, 45-32
- EnlsClass, 45-20, 45-30
- EnlsDeleteChar, 45-31
- EnlsDrawBox, 45-32
- EnlsDrawFormChars, 45-32
- EnlsDrawLine, 45-32
- EnlsFieldEdit, 26-32, 45-21, 45-31
- EnlsFieldEditByChar, 26-16, 26-32, 45-32
- EnlsFindC, 45-21, 45-32
- EnlsFindRC, 45-21, 45-32
- EnlsGetChar, 45-22, 45-31
- EnlsGetCharWidth, 45-31
- EnlsGetPrevChar, 45-31
- EnlsInsertChar, 45-32
- EnlsMapCharToStdValue, 45-20, 45-31
- EnlsMapStdValueToChar, 45-20, 45-31
- EnlsQueryBoxSize, 45-32
- EnterBootRom, 27-10
- EOI (*See* end-of-interrupt)
- ercOK, 4-3
- error
 - codes (*See* status codes)
 - handling, 5-6
 - message file, 26-20
 - reporting, 4-3
- error codes (*See* status codes)
- ErrorExit, 5-7, 33-11
- ErrorExitString, 5-7
- escape sequences, 11-49
- establishing
 - directory passwords, 12-12
 - communications channels, 40-15
 - interrupt handlers, 40-4
 - multiplexed interrupt handlers, 40-23
- event-driven, priority-ordered
 - process scheduling, 1-2, 1-3, 2-2
- events, 2-2
- examples of
 - CTOS calls, 4-3
 - naming conventions, 4-3
 - procedural interface, 4-2
- exceptions, 40-4 (*See also* internal interrupts)
- exchange routing, 30-49 to 30-50
- exchanges 2-2, 5-6
 - allocating, 30-18
 - defined, 30-11, 30-17
 - queues, 30-22
 - response, 30-16
 - sending messages to, 30-19
 - system service, 30-16
 - types, 30-18
 - waiting for messages at, 30-21
- exclusive access to resources, 31-5
- exclusive mode (mx), 18-10
- executable program, 5-1
- executing real mode applications, 3-13
- execution, thread of, 29-1
- Executive, 2-4, 5-5, 6-1, 10-3, 10-4
- exercising administrative control
 - over the cluster, 43-1, 44-3
- exit run files, 5-5 to 5-5
- Exit, 5-8, 33-11, 34-9
- ExitAndRemove, 5-8
- ExitListQuery, 39-22
- ExitListSet, 39-22
- expand down segment, 24-1

- expand up segment, 24-1
- ExpandAreaLL, 24-9, 24-14
- ExpandAreaSL, 24-9, 24-13
- ExpandDateTime, 26-22, 26-36
- expanded date/time format, 26-22, 27-4
- expanding boxes, 26-16
- expanding specifications, 30-42 to 30-43
- ExpandLocalMsg, 45-33
- explicitly enabling SCSI target mode functions, 18-21
- exports, 39-2
- extended address register, 41-5
- extended character sets, 11-3, 45-5
- extended native language support, 1-5
- extended partition descriptor, 27-4, 36-5
- extended process control block, 27-4
- extended system services, xxxviii, 28-3, 35-1 to 35-3
- extending memory, 4-14
- external interrupt handler
 - examples, 40-22
- external interrupt handling model, 40-5
- external interrupts, 40-2, 40-3
- extracting parameter data from ASCB, 6-3

F

- FAB (*See* file area blocks)
- far procedures, 38-7, 39-14
- FatalError, 5-7
- faulting pages into physical memory, 38-2

- faults, 40-26
- FCB (*See* file control blocks)
- FComparePointer, 26-28, 26-31
- features, operating system, xxxvii
- FHB (*See* file header blocks)
- FieldEdit, 26-32
- file access modes, 2-5
- file area blocks, 12-24, 12-34
- file control blocks, 12-24, 12-34
- file handles, 12-22, 12-24
- file header blocks, 12-6, 27-4, 12-23
- file management, 2-5
 - accessing files, 12-1
 - augmenting, 12-3
 - automatic volume recognition, 12-3
 - capabilities, 12-1
 - concurrent file access, 12-1
 - creating a file, 12-22
 - device independence, 12-21
 - hierarchical organization, 12-1
 - performing I/O to a disk file, 12-25
 - throughput capability of disk hardware, 12-1
 - using Nls.sys, 12-1
 - volume encryption, 12-3
- file password, 12-8, 12-13
 - adding passwords to files, 12-13
 - changing file passwords, 12-13
- file processor, 2-7
- file protection
 - assigning file passwords, 12-13
 - assigning volume passwords, 12-12
 - by password, 12-11
 - changing or removing directory passwords, 12-12
 - directory passwords, 12-12

- establishing directory passwords, 12-12
- volume encryption, 12-11
- volume passwords, 12-12
- file protection by protection level, 12-14 to 12-17
- file specifications
 - abbreviated, 12-10
 - default, 12-10
 - directory, 12-9
 - full file, 12-9
- files
 - access, 12-7
 - changing file length, 12-7
 - creating, 12-7
 - defined, 12-7
 - names, 12-8
 - protecting, 12-7
 - renaming, 12-7
- files, accessing using hybrid access methods, 20-5
- files, creating and accessing, 20-2
- FillBufferAsyncC, 15-4, 15-6
- FillBufferC, 15-6
- FillFrame, 10-18
- FillFrameRectangle, 10-18
- FilterDebugInterrupts, 27-11
- filtering
 - keyboard input, 11-15
 - ReadKbdInfo requests, 11-46
- filters, 2-4, 30-15, 33-8
 - defined, 33-22, 30-51
 - one-way pass-through, 33-22 to 33-23
 - replacement, 33-22
 - system requests for, 33-25
 - two-way pass-through, 33-24
 - uses, 33-25
- fixed length records, 20-2
- fixed partitions, 2-13
- fixed-length records, 20-1
- floating-point coprocessors, 40-3
- FlushBufferAsyncC, 15-6
- FlushBufferC, 15-6
- flushing cache entries, 25-12
- FMergedOs, 27-11
- Format Disk command, 13-4
- Format, 13-5
- FormatDateTime, 26-36
- FormatTime, 26-36
- FormatTimeDt, 26-36
- FormatTimeTm, 26-37
- formatting
 - cache memory, 25-6 to 25-8
 - characteristics, 19-1
 - disks, 13-4
- FormEdit, 26-32
- forms-oriented interfaces, 6-1
- ForwardRequest, 30-15, 30-55
- FP (*See* file processor)
- FProcessorSupportsProtectedMode, 27-11
- FProtectedMode, 27-11
- frame descriptor, 10-17, 27-4
- FrameBackSpace, 10-18
- frames, 3-1, 3-2
- FRmos, 27-11
- FRmosUser, 27-11
- FsCanon, 26-31
- FSrpUp, 44-4
- functions, 4-3
- FValidPbCb, 24-16
- FVSeries, 27-11

G

- GDT (*See* global descriptor table)
- general processor, 2-7
- generating a system, 43-1
- generating SAM (*See* SAMGen)
- Generic Print Access Method, 14-2, 19-1
- Generic Print System byte streams, 8-6
- Generic Print System, 14-1
- GenResString, 26-28, 26-43
- GetAltMsg, 45-26, 45-34
- GetAltMsgUnexpandedLength, 45-34
- GetBoardInfo, 32-18
- GetBsLfa, 9-2, 9-5
- GetCanonicalNodeAndVol, 12-48
- GetClstrGenerationNumber, 44-4
- GetClusterId, 26-42
- GetClusterStatus, 27-10
- GetClusterStatus, 44-1, 44-4
- GetCommLineDmaStatus, 16-6
- GetCoprocessorStatus, 27-11
- GetCParasOvlyZone, 38-16
- GetCtosDiskPartition, 13-4, 13-5
- GetDateTime, 26-22, 26-37
- GetDirInfo, 12-46
- GetDirStatus, 12-45
- GetErc, 26-20, 26-21
- GetErc, 26-35
- GetErcLength, 26-20, 26-35
- GetFhLongevity, 12-46
- GetFileErc, 26-20, 26-21, 26-35
- GetFileInfoByName, 12-44
- GetFileStatus, 12-44
- GetFRmosUser, 27-11
- GetKbdId, 11-56
- GetKeyboardId, 11-56
- GetLocalDaiNumber, 27-12
- GetModuleAddress, 41-2, 41-9
- GetModuleId, 41-9
- GetMsg, 45-33
- GetMsgUnexpanded, 45-33
- GetMsgUnexpandedLength, 45-33
- GetNlsDateName, 45-28
- GetNlsKeycapText, 45-28
- GetNlsTable, 45-17, 45-28
- GetOvlyStats, 38-16
- GetPartitionExchange, 36-8
- GetPartitionHandle, 36-8, 36-17, 36-20
- GetPartitionStatus, 27-12, 36-8, 36-16, 36-17, 36-20
- GetPartitionSwapMode, 36-21
- GetPAscb, 27-12
- GetPNlsTable, 45-28
- GetProcInfo, 32-3, 32-18
- GetPStructure, 11-33, 26-21, 27-9, 27-12, 36-17
- GetRsLfa, 22-3
- GetScsiInfo, 18-26
- GetSegmentLength, 24-16
- GetServerMsg, 45-26, 45-35
- GetSlotFromName, 32-18
- GetSlotInfo, 32-3, 32-18
- GetStamFileHeader, 20-8
- GetStandardErcMsg, 26-36
- GetSysCmdInfo, 26-18, 26-34
- GetUcb, 12-45
- GetUserFileEntry, 26-23, 26-23, 26-42
- GetUserNumber, 36-8, 36-20
- GetVhb, 12-49, 27-10

- global
 - descriptor table, 4-12, 5-6, 24-3
 - initialization, 39-2, 39-14, 39-20
 - linear address space, 24-3, 24-11, 33-9
 - linear addresses, 2-16, 3-2, 3-6
 - page map, 3-5
 - segments, 39-2, 39-9
 - thrashing, 3-2, 3-9
 - global descriptor table, 4-12, 5-6, 24-3
 - global segments, 39-2, 39-9
 - GP (See general processor)
 - GPAM (See Generic Print Access Method)
 - GPS (See Generic Print System)
 - grouping segments, 5-3
 - guidelines for writing
 - CMIHs, 40-18 to 40-19
 - CRIHs, 40-16 to 40-17
 - MIHs, 40-22
 - RIHs, 40-20 to 40-21
- ## H
- half- and full-duplex
 - communications, 15-3
 - handling commands, 26-18
 - handling error conditions, 36-1
 - handling nationalized strings, 26-15
 - hardware dependencies below byte stream interface level, 16-2
 - hash, 25-1, 25-4
 - heap, 24-1
 - hierarchical organization of files
 - node, 12-4
 - volume, 12-5
 - file specifications, 12-36
 - High Sierra directory record, 27-4
 - high-level interfaces, 4-8 to 4-9, 7-1, 7-3 to 7-4
 - high-resolution timing, 37-1
 - host adapter, 18-5
 - huge segments, 24-1, 24-4
 - defined, 4-10
 - managing 24-11
 - hypersegment swapping, 3-4
 - hypersegments, 2-12, 36-3 to 36-5, 36-8 (See also partition components)
 - code, 36-7
 - in-memory relationships, 36-3 to 36-5
 - local descriptor table, 36-5
 - long-lived memory, 36-7
 - short-lived memory, 36-7
 - user structure, 36-5 to 36-6
- ## I
- I/O
 - blocks, 12-34
 - interfaces, 4-8
 - nonoverlapped, 20-2, 20-4
 - overlapped, 22-1, 22-2
 - ICC (See inter-CPU communication)
 - ICC request/response ring queues, 32-2
 - ICC segments, 32-6
 - ICMS (See Intercontext Message Service)
 - identifying unique SCSI devices, 18-7
 - identifying workstations of a given hardware type, 26-27

- IDT (*See* interrupt descriptor table)
- image mode, 8-7
- import library, 39-2
- import, 39-2
- improving program performance, 38-14
- including delimiters in strings, 26-26
- index file, ISAM, 20-2
- Indexed Sequential Access Method, 12-3, 20-2 to 20-3, 21-1
- informing user of waiting mail, 26-28
- InitAltMsgFile, 45-26, 45-34
- InitCharMap, 10-21, 10-9
- InitCommLine, 16-2, 16-5, 40-15, 40-30
- InitErcFile, 26-20, 26-36
- initializing
 - cache, 25-5 to 25-8
 - character maps, 10-9
 - dynamic link libraries, 39-10
 - resource access, 26-8
 - timer request blocks, 37-3
- initiating video refresh, 10-9
- initiator, 18-20
- InitLargeOverlays, 38-16
- InitMsgFile, 45-33
- InitOverlays, 38-16
- InitSysCmds, 26-18, 26-35
- InitVidFrame, 10-9, 10-21
- input event types, storage of, 11-4
- installing trap handlers, 40-27
- InstallNet, 30-55
- InstallSystemCommon, 34-4
- instance initialization, 39-14, 39-2, 39-20
- instance segments, 39-2, 39-9
- INT, 40-25
- interboard routing (*See* inter-CPU communication)
- Intercontext Message Service, 30-5, 36-17
- Inter-CPU communication, 2-4, 32-1
- interface levels, defined, 4-8
- intermodule, general-purpose expansion bus (*See* X-Bus)
- internal interrupt handlers, 40-28
- internal interrupts, 40-2, 40-25 (*See also* traps)
- internationalization, 45-1
 - defined, 45-2
 - extended native language support, 1-5
 - keyboards, 1-4
- internationalizing software, 43-1
 - extending internationalization, 45-19
 - guidelines, 45-18
- interprocess communication, 2-2, 30-1, 30-52 to 30-53, 33-2, 34-1
- interrupt descriptor table, 40-2
- interrupt flag, 40-8
- interrupt handlers, 40-2
 - and general processor boards, 40-13
 - and virtual memory operating systems, 40-12
- communications, 40-15 to 40-19
- defined, 40-2
- guidelines for writing CMIHs, 40-18 to 40-19
- guidelines for writing CRIHs, 40-16 to 40-17
- guidelines for writing MIHs, 40-22

- guidelines for writing RIHs, 40-20
 - to 40-21
 - nesting, 40-13
 - packaging, 40-28 to 40-29
 - styles, 40-12, 40-14
 - interrupt handling styles, 40-1
 - interrupt hierarchy, 40-3
 - interrupt latency, 31-8
 - interrupt levels, 40-2, 40-3, 40-4
 - interrupt vector table, 40-2
 - interrupts
 - defined, 40-2
 - device, 40-3
 - external, 40-2, 40-3
 - hierarchy, 40-3, 40-4
 - internal, 40-2, 40-3 (*See also* traps)
 - interrupt handlers, 40-2
 - levels, 40-2
 - lost, 40-11
 - nonmaskable, 40-12
 - pending, 40-11
 - intersegment references, 5-4
 - IPC (*See* interprocess communication)
 - IPC applications
 - communicating between
 - application partitions, 30-5
 - communicating within an
 - application partition, 30-4
 - managing resources, 30-7
 - synchronizing processes, 30-6
 - IPC components, 30-11
 - IPC extension (*See* inter-CPU communication)
 - IPC messages (*See* request blocks)
 - IRET, 40-26
 - ISAM (*See* Indexed Sequential Access Method)
 - ISAM data set, 20-2 to 20-3, 21-1
 - ISAM.lib, 20-3
 - ISO directory record, 27-5
 - IVT (*See* interrupt vector table)
 - I-Bus
 - device management, 11-54
 - disconnecting drivers, 11-55
 - installing loadable drivers, 11-55
 - protocol, 11-55
 - reading input events, 11-55
 - I-Bus device management, 11-54 to 11-55
 - I-Bus style keyboard, 11-3
 - I-key, 11-3, 11-20
- ## K
- kernel functions, 2-1
 - kernel primitives, 4-4
 - kernel, 2-1
 - key post value, 11-9, 11-11
 - keyboard
 - application profile, 11-2, 11-43
 - configuration options, 11-43
 - I-Bus style, 11-3, 11-13
 - mapping IDs, 11-44
 - PC-style, 11-4, 11-13
 - setting options, 11-45
 - source, 11-31
 - supporting multiple profiles, 11-43
 - system profile, 11-4, 11-41
 - target, 11-31
 - emulation LEDs table, 11-31
 - keyboard and video independence, 11-54
 - keyboard byte streams, 8-8

- keyboard code, 11-3
- keyboard customization, 45-5
- keyboard data block, 11-3, 11-18 to 11-26
 - customizing, 11-31 to 11-36
 - general layout, 11-18
 - translation, 11-20
 - organization, 11-18 to 11-23
- keyboard event, 11-3
- keyboard hardware protocols, 11-13
- keyboard interrupt service routine, 11-9
- keyboard interrupt, 11-9
- keyboard management
 - features, 45-5 to 45-6
 - international keyboard support, 1-4
 - multiple keyboard support, 1-4
 - options, 11-1
 - overview, 45-7 to 11-10
- keyboard modes, 11-10 to 11-13
 - character, 11-2, 11-12
 - character plus, 11-12
 - comparing, 10-11
 - mapped unencoded, 11-12
 - raw unencoded, 11-9, 11-12
 - unencoded, 11-4, 11-11
 - unencoded plus, 11-12
- keyboard postprocessing, 11-10
- keyboard preprocessing, 11-9
- keyboard state, 11-10
- keyboard subtables, 10-16
- keyboard supporting tables, 11-18, 11-19, 11-22, 11-23
 - allowed states table, 11-23
 - allowed states, 11-23
 - chords table, 11-23
 - chords, 11-23
 - command masks table, 11-25

- conditions table, 11-23
- conditions, 11-23
- decoding table, 11-26
- defining toggle chords, 11-40
- diacritic key handling, 11-40
- diacritic masks table, 11-25
- multibyte masks table, 11-25
- multibyte strings, 11-40
- keyboard translation table, 11-22
- KeyboardProfile, 11-57
- keyboards
 - customizing, 1-5
 - source, 1-5
 - target, 1-5

L

- LaFromP, 24-16
- LaFromSn, 24-16
- language definition, 45-2
- large model, 24-6
- late binding (*See* dynamic linking)
- LDT (*See* local descriptor table)
- least-recently-used, 25-1
- level, 12-36
- lfa (*See* logical file address)
- LfsToMaster, 12-29
- LibFree, 39-22
- LibGetHandle, 39-22
- LibGetInfo, 39-22
- LibGetProcInfo, 39-22
- LibLoad, 39-22
- Librarian, 4-1
- line speed configuration, 44-1
- linear address space, 3-7, 36-9
- linear memory addresses, 4-13
- lines, 44-1 (*See also* cluster communication channels)
- linked-in service, 2-3

- Linker, 2-13, 4-1, 5-3
 - linking
 - dynamic, 4-5
 - object modules into run files, 5-1
 - to 5-3, 38-1
 - programs, 5-1 to 5-3
 - static, 4-5
 - linking, dynamic, 4-5
 - linking, static, 4-5
 - loadable request files, 33-7, 34-5
 - LoadBackgroundPalette, 10-22
 - LoadColorStyleRam, 10-22
 - LoadFontRam, 10-9, 10-21
 - loading additional run files into
 - partitions, 36-16
 - loading programs, 5-4 to 5-4, 36-16
 - LoadInteractiveTask, 36-12, 36-16, 36-23
 - LoadPrimaryTask, 36-16, 36-22
 - LoadRunFile, 36-18, 36-23
 - LoadTask, 36-18, 36-23
 - local descriptor table, 4-12, 36-3, 36-5, 38-8
 - local linear addresses, 2-17, 3-2, 3-6
 - local page map, 3-5
 - local resource-sharing networks
 - (See cluster)
 - local semaphores (See RAM semaphores)
 - local thrashing, 3-2, 3-9
 - localization, 45-1, 45-23
 - locating the CTOS disk partition, 13-4
 - locating the volume home block, 13-4
 - locked pages, 3-2
 - LockIn, 16-5
 - LockInContext, 36-21
 - locking a noncritical semaphore, 31-7
 - locking pages, 3-12
 - locking semaphores, 31-3
 - LockOut, 16-5
 - LockVideo, 10-23
 - LockVideoForModify, 10-23
 - LockXbis, 41-9
 - log file record format, 27-5
 - logical file address, 12-21
 - logical memory addresses
 - defined, 4-12
 - example, 4-12
 - logical unit number, 18-5, 18-7
 - long-lived memory, 2-22 to 2-23, 6-3, 24-10, 36-3, 36-7
 - long-lived, 24-2
 - LookUpField, 26-24, 26-41
 - LookUpNumber, 26-24, 26-41
 - LookUpReset, 26-24, 26-41
 - LookUpString, 26-24, 26-41
 - low memory allocation, 27-5
 - low resolution timing, 37-1
 - low-level interfaces, 4-8 to 4-9, 7-1
 - LRU (See least-recently used)
 - LUN (See logical unit number)
 - macros
 - using to distinguish between keys in the standard character set, 45-15
 - using to expand messages, 45-26
- ## M
- MakePermanent, 38-16
 - MakePermanentP, 38-17
 - MakeRecentlyUsed, 38-17

managing

- data packets, 18-22
- names, 26-18 to 26-20
- physical memory, 2-13, 2-19
- resources by system services, 28-2
- system date and time, 26-21

manipulating error messages, 26-20 to 26-21

MapBusAddress, 42-1, 42-3, 42-5, 42-8

MapCsIOvly, 38-17

MapIOvlyCs, 38-17

mapped unencoded, 11-9, 11-12

mapping

- addresses between clients and system services, 34-5
- device-independent operations to device-dependent, 9-1
- keyboard IDs, 11-44
- linktime to runtime addresses, 39-9

MapPStubPProc, 38-17

MapXBusWindow, 41-4, 41-9

MapXBusWindowLarge, 41-5, 41-9

maskable, 40-8

masking device priority levels, 40-10

master agent, 30-33

master file directory, 12-6

matching keyboards to data blocks, 11-42

McVersion, 26-38

mediated interrupt handlers, 40-13, 40-22

MediateIntHandler, 40-30

mediating buffer wait conditions, 32-6, 32-14 to 32-17

medium model, 24-6, 34-6

memory

- allocating for code and static data, 24-2
- long-lived, 24-10, 24-2
- partition, 24-2
- short-lived, 24-2

memory addresses

- logical memory addresses, 4-12
- physical, 4-10
- translating, 4-10 to 4-11

memory disk, 13-3, 25-3

memory management styles, xliii, 2-11 to 2-13

memory master, 41-3

memory organization

- application partition, 2-22 to 2-23
- system memory, 2-14 to 2-18
- application partition, 24-8 to 24-9

memory slave, 41-3

memory use, maximizing, 25-4

memory, short-lived, 5-6

MenuEdit, 26-32

merging requests, 33-7, 33-16

message file facility

- alternate message file operations, 45-26
- creating and editing message files, 45-24
- defined, 45-24
- standard message file operations, 45-26
- system service message file operations, 45-26
- using a single message file, 45-26
- using macros with messages, 45-26
- using multiple message files, 45-26
- using very few messages, 45-26

- message files, 43-1
- message work area, 26-20, 45-24
- messaged-based operation, 1-2, 1-3, 2-2
- messages, 30-23 (*See also* request blocks)
 - kernel primitives for receiving, 30-16 to 30-17
 - kernel primitives for sending, 30-12 to 30-15
- MIH (*See* mediated interrupt handlers)
- mode 3 DMA, 41-6
- Mode3DmaReload, 41-9
- models of computation, 5-3, 24-6, 34-6, 39-13
- module definition file, 39-2, 39-10
- MountVolume, 13-5
- Mouse Services operations
 - GetIBusDevInfo, 35-7
 - PDAssignMouse, 35-8
 - PDGetCursorPos, 35-7
 - PDGetCursorPosNSC, 35-7
 - PDInitialize, 35-6
 - PDLoadCursor, 35-7
 - PDLoadSystemCursor, 35-7
 - PDQueryControls, 35-7
 - PDQuerySystemControls, 35-7
 - PDReadCurrentCursor, 35-7
 - PDReadIconFile, 35-7
 - PDSetCharMapVirtual Coordinates, 35-6
 - PDSetControls, 35-8
 - PDSetCursorDisplay, 35-8
 - PDSetCursorPos, 35-8
 - PDSetCursorPosNSC, 35-8
 - PDSetCursorType, 35-6
 - PDSetMotionRectangle, 35-6
 - PDSetMotionRectangleNSC, 35-6
 - PDSetSystemControls, 35-8
 - PDSetTracking, 35-6
 - PDSetVirtualCoordinates, 35-6
 - PDTranslateNSCToVC, 35-8
 - PDTranslateVCToNSC, 35-8
 - ReadInputEvent, 35-7
 - ReadInputEventNSC, 35-7
- Mouse Services, 10-3, 11-14
- MouseVersion, 26-38
- MoveFrameRectangle, 10-18
- MoveOverlays, 38-16
- moving program segments between disk and memory, 38-1
- multibyte characters, 11-4, 45-1, 45-5
- multibyte strings, 11-4, 11-16
- multipartition memory
 - management, 2-12, 2-14 to 2-16, 3-3, 3-4, 4-13
- multipartition operating systems, 36-1, 36-9, 36-11, 38-1
- multiprogramming, 1-2, 1-3, 28-1, 28-2, 36-1, 36-6
- multitasking, 1-2, 1-4
- multithreading (*See* multitasking)
- multi-instance system services, 33-21
- mutual exclusion on CTOS, 31-10
- mutual exclusion semaphores, 31-3, 31-5
- mutual exclusion semaphores using Send and Wait, 31-11 to 31-12
- MWA (*See* message work area)

N

- name management, xli, 1-8
- named addresses, 39-6
- naming conventions, xlv to xlviii, 4-2 to 4-3
- nationalizable operations, 26-16
- nationalization, 1-2, 1-4, 45-5
- nationalizing programs, 4-15
- nationalizing the system date and time, 26-22
- native language support, 45-1 to 45-35
- native language support tables
 - accessing the functionality of, 45-6
 - character class, 45-12
 - Clustershare key post values, 45-15
 - ClusterShare keyboard extended codes, 45-15
 - Clustershare keyboard, 45-14
 - Clustershare video translation, 45-15
 - collating sequence, 45-10
 - Context Manager, 45-16
 - date and time formats, 45-9
 - date name translations, 45-10
 - file system case, 45-8
 - keyboard chords, 45-13
 - keyboard mapping, 45-7
 - keycap legends, 45-9
 - lowercase to uppercase, 45-8
 - multibyte escape keys, 45-14
 - NLS Strings, 45-14
 - number and currency formats, 45-10
 - source file, 45-2
 - special characters, 45-13
 - uppercase to lowercase, 45-8
 - video byte streams text, 45-8
 - yes or no strings, 45-13
- near procedure, 38-7
- nesting interrupt handlers, 40-10
- Net agent, 30-34, 30-51
- Net server, 30-34
- network configuration, 30-32
- network environment, 2-11
- network routing, 30-34, 30-45, 30-46 to 30-47
- network, 2-6
- networking, built-in, 1-2, 1-4
- NLS keyboard tables, 11-33
- NlsCase, 45-28
- NlsClass, 45-29
- NlsCollate, 45-29
- NlsFormatDateTime, 26-22, 26-37, 45-29
- NlsGetYesNoStrings, 26-15, 26-31, 45-29
- NlsGetYesNoStringSize, 26-15, 26-31, 45-29
- NlsKbd.sys, 1-5, 11-18, 11-19, 11-34 to 11-35, 45-5 (*See also* system keyboard file)
- NlsNumberAndCurrency, 45-29
- NlsParseTime, 26-22, 26-37, 45-29
- NlsSpecialCharacters, 45-29
- NlsStdFormatDateTime, 26-22, 26-37, 45-29
- NlsULCmpB, 26-15, 26-31, 45-30
- NlsVerifySignatures, 45-30
- NlsYesNoOrBlank, 26-15, 26-31, 45-5, 45-30
- NlsYesNoStrings, 45-5
- NlsYesNoStringSize, 45-5
- NlsYesOrNo, 26-15, 26-31, 45-30, 45-5

node names, reserved, 30-42
node, defined, 30-32
noncritical semaphores, 31-3, 31-6,
31-7
nonterminatable semaphores, 31-3,
31-10
normal mode, 8-7, 11-46
NPrint, 26-17, 26-33
null process, 29-3, 29-4

O

object module procedures, 4-4
ObtainAccessInfo, 35-6
obtaining
 cache entries, 25-9
 cache statistics, 25-13
 cache status, 25-12
 cluster status, 44-1
 command information, 26-18
 partition status, 36-17
 physical memory addresses, 42-7
 system information, 27-9
 system data and time, 26-21 to
 26-22
ObtainUserAccessInfo, 35-6
offset, 4-12, 24-3 to 24-4 (*See also*
 relative address)
OpenByteStream, 8-15
OpenByteStreamC, 15-5, 15-6
OpenDaFile, 23-4
OpenFile, 12-8, 12-24, 12-43
OpenFileLL, 12-22, 12-24, 12-46
opening
 command files, 26-18
 files, 12-24
 system semaphores, 31-6 to 31-6
 byte streams, 8-3
OpenNlsFile, 45-30
OpenRsFile, 22-3
OpenRtClock, 37-3, 37-8
OpenServerMsgFile, 45-26, 45-35
OpenUserFile, 26-24, 26-41
OpenVidFilter, 8-4, 9-5
operating system features, xxxvii
operating system flags, 27-5
operating system types, 2-7 to 2-10
operational modes, 1-1
operations
 kernel primitives, 4-4, 4-6
 object module procedures, 4-4
 object module procedures, 4-5
 request-based, 4-7
 system-common procedures, 4-5
 using the request procedural
 interface to system services,
 4-4
optimizing
 a cache, 25-10
 performance, 3-11, 25-13
organizing parameter data in
 ASCB, 6-3
OsVersion, 4-15, 27-12
OutputBytesWithWrap, 26-17,
 26-33
OutputQuad, 26-17, 26-33
OutputToVid0, 8-15
OutputWord, 26-17, 26-33
outstanding requests, 30-32
overlay descriptors, 38-6
overlay zone header, 38-6
overlays, 38-2 (*See also* virtual code
 management)
overflow, 40-11
oversubscribing memory, 2-21, 3-2,
 3-3, 3-8

P

- pages, 3-1, 3-3
 - cleaning, 3-3
 - faults, 40-26
 - mapping, 3-5 to 3-5
 - replacement, 3-9 to 3-11
- paging parameters, 3-14
- paging service, 2-13, 2-16
- paging service, components, 3-12
- paragraph, 24-3, 24-9
- parallel port interrupt handlers, 40-22
- parameters, 6-2
- ParseFileSpec, 12-37 to 12-38, 12-47
- ParseSpecForDir, 12-47
- ParseSpecForFile, 12-47
- ParseSpecForNode, 12-47
- ParseSpecForPassword, 12-47
- ParseSpecForVol, 12-47
- ParseTime, 26-37
- parsing
 - answers to Executive yes/no options, 26-15
 - configuration files, 26-23 to 26-26
 - device/file specifications, 8-14
 - nonstandard configuration files, 26-25
 - specifications, 16-2
 - standard configuration files, 26-24 to 26-25
- partition
 - components, 33-11
 - defined, 5-1
 - configuration block, 27-5, 36-5
 - descriptor, 27-5
 - handle (See user number)
 - information block, 27-5
 - keyboard information structure, 27-5
 - partition configuration block, 27-5, 36-5
 - partition descriptor, 27-5
 - partition handle (See user number)
 - partition information block, 27-5
 - partition keyboard information structure, 27-5
 - partition managing programs, 28-3, 36-1, 36-12
 - partition memory, allocating and deallocating, 24-9 to 24-10 (See also short-lived and long-lived memory)
 - partition, 28-3
 - variable length parameter block structure, 6-3, 6-4
 - swapping, 3-4, 36-13 to 36-14
 - partitioned memory, 36-1
- partitions, 2-11
 - application, 36-2
 - components, 36-3 to 36-7
 - disk, 13-4
 - fixed, 36-2
 - system, 36-2
 - types, 36-2
 - variable, 36-2, 36-7
 - with multiple run files, 36-18
- passwords
 - device, 12-8
 - directory, 12-8
 - file, 12-8
 - specifying, 12-8
 - types, 12-8
 - volume, 12-8

- password protection
 - encrypted, 2-5
 - protection level, 2-5
- password protection to SCSI
 - devices, 18-5
- path handle, 18-9
- patience, 40-9, 40-27
- PC emulation, 1-1
- PDGetCursorPos, 35-7
- peek mode (mp), 18-10
- performance, optimizing I/O to
 - sequential access devices, 8-10
- Performance Statistics structure, 27-6
- Performance Statistics System
 - Service operations
 - ClosePsSession, 35-9
 - DeinstPsServer, 35-9
 - GetPsCounters, 35-9
 - OpenPsLogSession, 35-9
 - OpenPsStatSession, 35-9
 - PsCloseSession, 35-9
 - PsDeinstServer, 35-9
 - PsGetCounters, 35-9
 - PsOpenLogSession, 35-9
 - PsOpenStatSession, 35-9
 - PsReadLog, 35-10
 - PsResetCounters, 35-10
 - ReadPsLog, 35-10
 - ResetPsCounters, 35-10
- performance, 40-12
- periodically checking for messages, 30-32
- physical address space, 4-10
- physical devices, directly
 - controlling, 7-3
- physical memory address, defined, 4-13, 4-10
- physical records, 20-6
- piecemealing DMA buffers, 42-7
- PIT (*See* programmable interval timer)
- pixels, 10-1
- placing
 - clients in wait state, 30-2
 - error messages in user-supplied byte streams, 26-21
 - information in video control blocks, 10-9
- playback mode, 11-47
- polling, 44-2
- port structure, 27-6
- porting applications to CTOS, 31-2
- porting programs to different native languages, 4-15
- PosFrameCursor, 10-18
- posting data blocks, 11-17, 11-36, 11-42, 11-54
- PostKbdTable, 11-36, 11-57
- predefined byte stream work areas, 8-4
- prefaulting pages, 3-3, 3-11
- preopened byte streams, 8-3
- pre-GPS spooler byte streams, 8-7
- primary partition, 33-11, 33-20
- primary task, 36-18, 38-15
- PrintAltMsg, 45-34
- printer byte streams, 8-5
- PrintErc, 26-20, 26-21, 26-36
- PrintFileClose, 26-17, 26-33
- PrintFileOpen, 26-17, 26-33
- PrintFileStatus, 26-17, 26-33
- printing modes, 8-7
- PrintMsg, 45-33
- prioritizing interrupt signals, 40-9
- procedural interface, 4-2

- process, 2-1, 28-2
 - context switch, 29-3
 - context, 29-2
 - defined, 29-1
 - preempting execution, 29-4
 - priorities, 29-2, 29-3, 30-53
 - scheduling, 29-2
 - states, 29-4
- process control block, 27-6, 29-3
- process priorities, 29-2, 29-3, 30-53
- process, preempting execution of, 29-4
- processes
 - client, 2-3
 - system service, 2-3
- processor modes, changing 1-6
- ProcInfoNonres, 38-7
- ProcInfoRes, 38-6
- producer/consumer model
 - using CTOS kernel primitives, 31-15 to 31-16
 - using semaphores, 31-13 to 31-15
- program, 36-18
 - exceptions, 40-25
 - performance, 11-16, 25-3, 38-14
 - portability, 26-16
 - termination, 5-5 to 5-5, 11-54
- ProgramColorMapper, 10-22
- programmable interrupt controller, 40-9
- programmable interval timer, 28-3, 37-1, 37-6
- programmatic interface, 4-2
- programming documentation, 4-1
- programming tools (*See also* development utilities)
 - Linker, 24-6, 24-9
 - Resource Compiler, 26-3
 - Resource Librarian, 26-3
- programs and run files, 2-10 to 2-11
- prompts, 6-2
- protected mode addressing, 4-10, 4-13
- protected mode advantages, 4-14
- protected mode enhancements
 - caching, 1-5
 - extended native language support, 1-5
 - multiple and international keyboard support, 1-5
 - protected mode operation, 1-6
 - real mode application support, 1-6
 - SCSI management, 1-6
 - variable partitions with code sharing, 1-7
- protected mode operation, 1-4
- protected mode programs, 5-4
- protection by protection level, 12-14 to 12-17
- protection levels, 12-14 to 12-17
- PSend, 30-55, 40-5, 40-30
- pseudointerrupt, 40-24
- public procedures, 38-6
- PurgeMcr, 11-59
- PutBackByte, 9-2, 9-5
- PutByte, 26-17, 26-34
- PutChar, 26-17, 26-34
- PutCharsAndAttrs, 10-19
- PutFrameChars, 10-19
- PutFrameCharsAndAttrs, 10-19
- PutPointer, 26-17, 26-34
- PutQuad, 26-17, 26-34
- PutWord, 26-17, 26-34

Q

- QIC (*See* quarter-inch cartridge)
- quarter-inch cartridge, 8-9
- QueryBigMemAvail, 24-9, 24-15
- QueryBoardInfo, 32-18
- QueryBounds, 10-19
- QueryCharsAndAttrs, 10-19
- QueryCoproprocessor, 27-12
- QueryCursor, 10-19
- QueryDaLastRecord, 23-4
- QueryDaRecordStatus, 23-4
- QueryDcb, 27-11
- QueryDefaultRespExch, 30-54
- QueryDeviceName, 13-5
- QueryDeviceNames, 13-5
- QueryDiskGeometry, 13-5
- QueryExitRunFile, 5-8
- QueryFrameAttrs, 10-20
- QueryFrameBounds, 10-19
- QueryFrameChar, 10-20
- QueryFrameCharsAndAttrs, 10-20
- QueryFrameCursor, 10-20
- QueryFrameString, 10-20
- querying
 - communications parameters, 15-5
 - paging statistics, 3-13
 - parameter data in ASCB, 6-4
- QueryIOOwner, 24-15
- QueryKbdLeds, 11-56
- QueryKbdState, 11-58
- QueryLdtR, 27-12
- QueryMail, 26-28, 26-43
- QueryMemAvail, 24-9, 24-15
- QueryModulePosition, 41-9
- QueryNodeName, 33-28
- QueryPagingStatistics, 3-13, 3-15
- QueryRequestInfo, 33-8, 33-10, 33-28
- QueryTrapHandler, 40-28, 40-30
- QueryVidBs, 9-2, 9-5, 10-8
- QueryVideo, 10-21
- QueryVidHdw, 10-9, 10-20
- QueryWsNum, 44-4
- QueryZoomBoxPosition, 26-32
- QueryZoomBoxSize, 26-33
- queue entry header, 27-6
- queue file header, 27-6
- Queue Manager operations
 - AddQueue, 35-10
 - AddQueueEntry, 35-10
 - CleanQueue, 35-10
 - DeinstallQueueManager, 35-10
 - EstablishQueueServer, 35-10
 - GetQmStatus, 35-10
 - MarkKeyedQueueEntry, 35-11
 - MarkNextQueueEntry, 35-11
 - ReadKeyedQueueEntry, 35-11
 - ReadNextQueueEntry, 35-11
 - RemoveKeyedQueueEntry, 35-11
 - RemoveMarkedQueueEntry, 35-11
 - RemoveQueue, 35-11
 - RescheduleMarkedQueueEntry, 35-11
 - RewriteMarkedQueueEntry, 35-11
 - TerminateQueueServer, 35-12
 - UnmarkQueueEntry, 35-12
- queue status block, 27-6
- QueueMgrVersion, 26-38
- queuing messages at exchanges, 30-22

R

- RA (*See* offset; *See also* relative address)
- RAM semaphores, precautions, 31-18 to 31-19
- RAM semaphores, 31-3, 31-4
- random I/O, 20-2, 20-4
- raw interrupt handlers, 40-12
- raw unencoded mode, 11-4
- raw unencoded value, 11-9
- raw unencoded, 11-12, 11-13, 11-16
- RDT (*See* resource descriptor table)
- Read, 12-43
- ReadActionCode, 11-52, 11-58
- ReadActionKbd, 11-52, 11-58
- ReadAsync, 12-49, 30-30
- ReadBsRecord, 8-15
- ReadByte, 8-15
- ReadBytes, 8-16
- ReadByteStreamParameterC, 15-7
- ReadCommLineStatus, 16-4, 16-5, 40-30
- ReadDaFragment, 23-2, 23-5
- ReadDaRecord, 23-4
- ReadDirSector, 12-45
- ReadHardId, 26-27, 26-43
- reading and writing a file, 12-25 to 12-26
- reading data blocks
 - comparing reading methods, 11-39
 - in an object module, 2-34
 - into application memory, 11-36 to 11-38
 - reading binary file into local memory, 11-38
 - using byte streams, 11-37
 - using ReadOsKbdTable, 11-37
- reading keyboards, 11-14
- reading submit files, 11-14, 11-48
- ReadInputEvent, 11-14
- ReadKbd, 11-56
- ReadKbdDataDirect, 11-56
- ReadKbdDirect, 11-49, 11-57
- ReadKbdInfo, 11-12, 11-14 to 11-16, 11-55, 11-57, 45-6
- ReadKeySwitch, 32-18
- ReadMcr, 11-59
- ReadOsKbdTable, 11-37, 11-58
- ReadRsRecord, 22-3
- ReadStatusC, 15-7, 16-4
- ReadToNextField, 26-24, 26-25, 26-41
- ready state, 29-4
- read-only SCSI path parameters, 18-9
- real mode addressing, 4-13
- real mode application support, 1-4, 1-6
- real mode applications, 4-10
- ReallocHugeMemory, 24-11, 24-12, 24-17
- realtime clock, 28-3, 37-1
- ReceiveCommLineDma, 16-6
- receiving
 - messages, 30-16 to 30-17
 - requests, 32-11
 - responses, 32-12
- record fragment, 23-2
- record sequential access method, 12-3, 20-3, 22-1
- record sequential work area, 22-2
- recording file, 11-49, 11-54
- recording mode, 11-49

- records
 - blocked, 20-1, 23-1
 - fixed length, 20-1, 20-2, 23-1
 - fragment, 23-2
 - number, 23-1
 - physical, 20-6
 - random access to, 23-1
 - sequential access to, 22-1
 - spanned, 20-1, 23-1
 - standard file header, 20-6
 - standard record trailer, 20-6
 - variable length, 20-1, 22-1
- redefining keys, 11-30
- redirecting video byte streams, 5-4, 10-5
- redo keystroke buffer, 6-5
- reducing disk access, 26-4
- reentrant code, 34-6, 34-8, 39-5
- referencing memory
 - physical memory, 4-14
 - static memory allocated to other programs, 4-14
- register, BR0, 40-13
- ReinitLargeOverlays, 38-16
- ReinitOverlays, 38-16
- ReinitStubs, 38-17
- related documentation, xliii to xlv
- relative address, 4-12, 24-3 to 24-4
 - (*See also* offset)
- release documentation, 43-1
- ReleaseByteStream, 8-16
- ReleaseByteStreamC, 15-8
- ReleasePermanence, 38-17
- ReleaseRsFile, 22-3
- releasing cache entries, 25-11 to 25-12
- RemakeAliasForServer, 24-16
- RemakeFh, 12-46
- RemapBusAddress, 42-8
- RemoteBoot, 32-18
- RemoveFfsBrackets, 12-48
- RemovePartition, 36-8, 36-16, 36-17, 36-23
- removing interrupt handlers, 40-19
- removing partitions, 36-17
- RenameByteStream, 9-5
- RenameFile, 12-7, 12-43
- ReopenFile, 12-46
- repeat attributes table, 27-6
- repetitive timing, 37-5
- replacing local trap handlers, 40-28
- replacing pages, 3-8, 3-9
- representing the system date and time in compact form, 26-21
- Request, events that occur when called, 30-45
- request block routing code, 30-43
- request blocks, 30-1, 30-2, 30-11, 30-12, 33-2
 - components, 30-24
 - constructing, 30-30
 - control information, 30-26
 - example of Write request, 30-28 to 30-29
 - format, 30-24
 - header, 30-25 to 30-26
 - request data item, 30-27
 - response data item, 30-27
 - routing code, 30-27
- request codes, 33-2
 - assigning to system service requests, 30-9
 - defining system requests, 30-10
 - levels, 30-9
 - linked into operating system, 30-10

- registering, 30-9
- reserved, 30-9
- using to route requests, 30-9
- request definition, 33-7
- request procedural interface, 2-3, 30-1 to 30-4, 30-53, 33-4
 - defined, 4-6
 - using to access system services, 4-7
- request routing
 - across the cluster, 44-3
 - across the network, 30-46 to 30-47
 - by handle, 32-4
 - by specification, 32-4
 - defining for SRPs, 32-3 to 32-5
 - local exchange routing, 30-49 to 30-50
 - on SRPs, 30-49
- request routing across the cluster, 44-3
- request routing table, 30-45, 30-53
- request routing table, extending, 33-6
- Request, 30-2, 30-12, 30-53, 30-54, 32-11
- Request.sys, 33-7, 33-16
- RequestDirect, 30-15, 30-55, 32-11
- RequestRemote, 32-4, 32-18
- requests
 - global, 33-14
 - guidelines for defining, 33-14
 - local, 33-14
 - routing by handle, 30-34 to 30-41
 - routing by specification, 30-41 to 30-44
 - system, 33-14, 33-17 to 33-20
- RequestTemplate.txt, 33-14, 33-15, 33-16
- request-based operations, 4-7
- request-based system services, 28-2
- ReserveBusAddress, 42-8
- ReservePartitionMemory, 36-22
- reserving system-common numbers, 34-10
- ResetCommLine, 16-3, 16-5
- ResetDeviceHandler, 40-19, 40-30
- ResetFrame, 10-20
- ResetIbusHandler, 11-55, 11-59
- ResetMemoryLL, 6-8, 24-10, 24-9, 24-14
- ResetTimerInt, 37-7, 37-8, 40-31
- ResetTrapHandler, 40-28, 40-31
- ResetVideo, 10-9, 10-21
- ResetVideoGraphics, 10-21
- ResetXBusMIsr, 40-31, 41-10
- ResizeIOMap, 24-15
- ResizeSegment, 24-11, 24-17
- resizing segments, 24-12
- resource descriptor table, 26-4
- resource descriptor, 26-4
- resource handles
 - defined, 30-34
 - interpreting the bits in, 30-35 to 30-41
- resource ID, 26-4
- resource management, xli
- resource sharing, 18-4, 30-34
- resource type codes, 26-4
- resource work area, 26-4
- resources, defined, 26-2
- resources, system, 33-1
- Respond, 30-3, 30-14, 30-53, 30-54
- response exchange, 30-18
- restarting instructions, 40-26
- retrieving error message text, 26-21
- retrofitting overlay programs, 38-2

- return overlay descriptors, 38-7
 - ReuseAlias, 24-16
 - ReuseAliasLarge, 24-16
 - RgParam, 6-4, 6-6, 6-10
 - RgParamInit, 6-8, 6-10
 - RgParamSetSimple, 6-10
 - RIH (*See* raw interrupt handlers)
 - RkvsVersion, 26-39
 - RMOS (*See* running real mode applications)
 - ROD (*See* return overlay descriptors)
 - roll call, 44-2
 - routing by handle, 30-34 to 30-41
 - routing by specification, 30-41 to 30-44
 - routing code
 - values for expanding specifications, 30-44
 - values for routing requests, 30-44
 - routing request-based services, 34-5
 - RSAM (*See* record sequential access method)
 - RSAM buffer, 22-2
 - RsrcCopyFromRsrcSet, 26-11
 - RsrcEndSetAccess, 26-10
 - RsrcGetCountAllSetType, 26-12
 - RsrcGetCountSetType, 26-12
 - RsrcInitSession, 26-8
 - RsrcInitSetAccess, 26-11 to 26-12
 - RsrcSessionEnd, 26-8
 - RsrcSessionInit, 26-7
 - RSWA (*See* record sequential work area)
 - RS-232-C, 2-9, 8-8, 16-2
 - RS-422, 2-6, 2-9
 - RS-485, 2-6, 2-9
 - RTC (*See* realtime clock)
 - run files, 5-1, 5-3, 6-5, 24-6
 - run queues, 30-53, 31-7
 - running applications larger than available memory, 38-1
 - running real mode applications, 1-6
 - running state, 29-4
 - runtime dynamic linking, 39-20 to 39-21
 - RWA (*See* resource work area)
- ## S
- SA (*See* segment address)
 - SAM (*See* sequential access method)
 - SAMGen, 8-2
 - SbPrint, 26-17
 - SbPrint, 26-34
 - ScanToGoodRsRecord, 22-3
 - scheduler, 29-3
 - scheduling techniques
 - event-driven priority ordered, 29-2
 - time-based, 29-2
 - scratch volume, 12-6
 - screen attributes, 10-2
 - ScrollFrame, 10-20
 - SCSI (target) ID, 18-5, 18-7
 - SCSI access modes
 - exclusive mode (mx), 18-10
 - peek mode (mp), 18-10
 - shared mode (ms), 18-10
 - SCSI bus, 18-2
 - SCSI command structure
 - information transfer phases, 18-14 to 18-15
 - order of commands, 18-15
 - types of commands, 18-14
 - using to control devices, 18-16
 - SCSI device, 18-7

- SCSI error conditions, 18-16
- SCSI Manager
 - SCSI bus, 18-2
 - as SCSI initiator, 18-20
 - configuring a SCSI Manager, 18-7
 - electrical layer, 18-1
 - logical session layer, 18-2
 - overview, 18-4
 - physical layer, 18-1
 - relationship to devices, user programs, 18-5
 - relationship to the file system, 18-12 to 18-13
 - target mode, 18-20
 - transport protocol layer, 18-1
- SCSI manager target mode, 18-20 to 18-21
- SCSI passwords, 18-11
- SCSI path, 18-5 to 18-6
- SCSI peripheral device, 18-1
- SCSI sense data, 18-17 to 18-20
- SCSI target mode commands, 18-21
- SCSI (Small Computer Systems Interface), 18-1 to 18-29
 - ScsiCdbDataIn, 18-16, 18-27
 - ScsiCdbDataInAsync, 18-27
 - ScsiCdbDataOut, 18-16, 18-27
 - ScsiCdbDataOutAsync, 18-27
 - ScsiClosePath, 18-26, 18-9
 - ScsiManagerNameQuery, 18-26
 - ScsiOpenPath, 18-5, 18-26
 - ScsiQueryInfo, 18-26
 - ScsiQueryPathParameters, 18-9, 18-26
 - ScsiRequestSense, 18-27
 - ScsiReset, 18-27
 - ScsiSetPathParameters, 18-9, 18-26
 - ScsiTargetCdbCheck, 18-28
 - ScsiTargetCdbWait, 18-28
 - ScsiTargetDataReceive, 18-22, 18-28
 - ScsiTargetDataReceiveAsync, 18-28
 - ScsiTargetDataTransmit, 18-22, 18-28
 - ScsiTargetDataTransmitAsync, 18-28
 - ScsiTargetOperationsAbort, 18-29
 - ScsiWaitCdbAsync, 18-27
 - ScsiWaitTargetDataAsync, 18-29
- secondary task, 36-18, 38-15
- segment address, 4-12, 24-2, 24-3
- segment attributes, 39-10
- segment base address, 4-12
- segment descriptors, 4-12, 24-4
- segment elements, 5-3
- segment not-present, 40-26
- segment offset, 24-2
- segment, expand down, 24-1
- segment, expand up, 24-1
- segment, huge, 24-1
- segmentation models, 5-3
- segmented addressing, 24-3
- segments, 24-2
 - address, 4-12, 24-2, 24-3
 - code, 5-3, 24-4, 24-6
 - data, 24-4
 - defined, 4-10, 24-3
 - dynamic data, 24-5, 24-7
 - dynamic link library, 39-19
 - global, 39-9
 - huge, 24-4
 - instance, 39-9
 - limit, 24-4
 - nonshared, 39-11 (*See also* global segments)
 - offset, 24-3 to 24-4

- shared, 39-12 (*See also* instance segments)
- special use of dynamic link library instance segments, 39-20
- static data, 5-3, 24-5, 24-6, 24-9
- types, 24-4
- selecting
 - file access methods, 20-6 to 20-7
 - keyboard translation tables, 11-23
 - nationalizable operations, 4-15
 - system services to write, 34-7
 - W-, X-, and Z-block buffers, 32-9
- selectively masking devices, 40-10
- self-exiting applications, 36-1
- self-loading applications, 36-1
- semaphores, xlii, 1-8, 39-13
 - defined, 31-1
 - handle, 31-1, 31-3
 - lock bit, 31-1, 31-5
 - owner, 31-3, 31-5
 - variable, 31-3, 31-17
- SemClear, 31-7, 31-8, 31-20, 31-21
- SemClearCritical, 31-9, 31-21
- SemClose, 31-5, 31-20
- SemLock, 31-7, 31-20
- SemLockCritical, 31-9, 31-21
- SemMuxWait, 31-13, 31-21
- SemNotify, 31-13
- SemOpen, 31-5, 31-20
- SemSet, 31-12, 31-21
- SemWait, 31-12, 31-21
- Send, 30-15, 30-55
- SendBreakC, 15-7
- sending a request, 32-8 to 32-9
- sending a response, 32-11
- sending and receiving messages, 32-12 to 32-14
- sending data to the video and a second device, 26-17
- sending messages, 2-2, 30-12 to 30-15
- SeqAccessCheckpoint, 35-13
- SeqAccessClose, 35-12
- SeqAccessCtrl, 35-12
- SeqAccessDiscardBufferData, 35-13
- SeqAccessModeQuery, 35-13
- SeqAccessModeSet, 35-13
- SeqAccessOpen, 35-12
- SeqAccessRead, 35-12
- SeqAccessRecoverBufferData, 35-13
- SeqAccessStatus, 35-12
- SeqAccessWrite, 35-12
- sequential access byte streams, 8-9
- sequential access method
 - advantages, 8-1
 - customizing, 8-2
- Sequential Access Service operations
 - SeqAccessCheckpoint, 35-13
 - SeqAccessClose, 35-12
 - SeqAccessCtrl, 35-12
 - SeqAccessDiscardBufferData, 35-13
 - SeqAccessModeQuery, 35-13
 - SeqAccessModeSet, 35-13
 - SeqAccessOpen, 35-12
 - SeqAccessRead, 35-12
 - SeqAccessRecoverBufferData, 35-13
 - SeqAccessStatus, 35-12
 - SeqAccessWrite, 35-12
- serial communications channels, 8-8
- serial port interfaces, 16-1
- SerialNumberOldOsQuery, 27-12
- SerialNumberQuery, 27-13

- server workstation, 2-7, 2-8
- server, 2-6, 2-7, 30-1, 44-1, 44-2
- ServerRq, 30-53, 33-8, 33-10, 33-28, 34-4, 34-8
- service exchange, 2-2, 30-3, 30-18, 30-45, 30-53, 33-3, 30-53
- Set386TrapHandler, 40-27, 40-31
- SetAlphaColorDefault, 10-22
- SetBsLfa, 9-2, 9-5
- SetDaBufferMode, 23-5
- SetDateTime, 26-22, 26-37
- SetDateTimeMode, 26-37
- SetDefault386TrapHandler, 40-28, 40-31
- SetDefaultTrapHandler, 40-28, 40-31
- SetDeviceHandler, 40-19, 40-31
- SetDevParams, 13-5
- SetDirStatus, 12-46
- SetDiskGeometry, 13-5
- SetExitRunFile, 5-8
- SetFhLongevity, 12-22, 12-46
- SetField, 26-24, 26-25, 26-42
- SetFieldNumber, 26-24, 26-42
- SetFileStatus, 12-44
- SetIBusHandler, 11-55, 11-59
- SetImageMode, 9-2, 9-4
- SetImageModeC, 15-7
- SetIntHandler, 40-19, 40-28, 40-32
- SetIOOwner, 24-15
- SetKbdLed, 11-57
- SetKbdUnencoded, 11-12
- SetKbdUnencodedMode, 11-48, 11-58
- SetKeyboardOptions, 11-45, 11-58
- SetLdtRDs, 40-32
- SetLpIsr, 40-22, 40-32
- SetMsgRet, 5-8, 33-10
- SetNode, 12-45
- SetPartitionLock, 36-23
- SetPartitionName, 33-11, 33-28, 36-20
- SetPartitionSwapMode, 36-21
- SetPath, 12-10, 12-45
- SetPrefix, 12-10, 12-45
- SetRsLfa, 22-3
- SetScreenVidAttr, 10-9, 10-21
- SetSegmentAccess, 24-16
- SetStyleRam, 10-22
- SetStyleRamEntry, 10-22
- SetSwapDisable, 36-21
- SetSysInMode, 11-46, 11-48, 11-58
- SetTimerInt, 37-6, 37-8, 40-32
- setting
 - keyboard options, 11-45
 - reverse video or half-bright, 10-9
 - semaphores (See locking semaphores)
- setting up
 - a base RWA, 26-12
 - DLL search paths, 39-17
 - operating environments, 2-10
- SetTrapHandler, 40-27, 40-32
- SetUserFileEntry, 26-23, 26-7, 26-42
- SetVideoTimeout, 10-21
- SetXBusMIsr, 40-23, 40-32, 41-10
- SgFromSa, 24-17
- shared mode (ms), 18-10
- shared resource processor board
 - features, 2-9 to 2-10
- shared resource processor boards
 - cluster processor, 17-1
 - general processor, 2-7
 - storage processor, 2-7
 - terminal processor, 17-1
- shared resource processor, 2-4
- sharing code and data, 2-24

- sharing server run files, 12-29
- ShortDelay, 37-3, 37-8
- short-lived memory, 2-22, 5-6, 36-3, 36-7
- short-lived, 24-2
- ShrinkAreaLL, 24-9, 24-14
- ShrinkAreaSL, 24-9, 24-14
- ShrinkPartition, 24-15
- signaling and synchronizing processes, 31-12
- signaling semaphores, equivalent CTOS functions, 31-15
- signaling semaphores, 31-3, 31-6
- SignOn password, 12-13
- SignOnLog, 26-43
- sizing
 - buffers, 23-2
 - programs, 5-3
- slot numbering, 32-2
- Small Computer Systems Interface (SCSI) management, 1-4, 1-6, 18-1 to 18-29
- small model, 24-6
- SnFromSr, 24-17
- software foundation, 1-2
- software interrupts, 40-25
- source file, 45-2
- source keyboard, 11-31
- SP (*See* storage processor)
- spanned records, 20-1
- special keys table, 27-6
- specification routing rules, 30-41
- specifying
 - default file passwords, 12-9
 - local file systems, 12-29
 - passwords, 12-8
 - screen coordinates and frame dimensions, 10-9
 - SCSI path parameters, 18-8
 - window sizes, 41-4
- Spooler operations
 - ConfigureSpooler, 35-14
 - SpoolerPassword, 35-14
- Spooler queue entries, 2-6
- SpoolerVersion, 26-39
- SrFromSn, 24-17
- SRP
 - name table, 32-4
 - request routing, 30-47 to 30-49
 - routing types, 30-49, 32-3 to 32-5
 - terminal management, 17-1
- stack, 39-4
- standard character set, 11-4
- standard file header, 20-6, 27-6
- standard operating system libraries, xxxviii
- standard record header, 23-1, 27-7
- standard record trailer, 20-6, 23-1
- starting a resource session, 26-7
- static data segment, 24-5, 24-6
- statically linked object modules, 39-4
- StaticsDesc, 38-6
- StatisticsVersion, 26-39
- status codes, 4-3, 4-7
- sticky keys, 11-45
- storage processor, 2-7
- storing
 - command form parameters, 6-5
 - parameter data, 6-3
- string
 - delimiters, 26-26
 - masks table, 27-7
 - offsets table, 27-7
 - table, 27-7
- string operations
 - comparing strings, 26-14
 - handling nationalized strings, 26-15
- strings table, 27-7

- StringsEqual, 26-15, 26-31
- structured file access methods, 20-1
- stub, 38-8
- stubs array, 38-6
- submit files, 11-54
 - editing precautions, 11-50
 - entering user data, 11-51
 - escape sequence, 11-49 to 11-51
 - playback, 11-47
 - recording, 11-49
- submit mode, 11-47
- subparameters, 6-2 to 6-3, 6-4
- SuperGen Series 5000, 41-1
- suppressing duplication of volume control structures, 2-5
- suspended state, 29-5
- swap files, 36-14
- SwapInContext, 36-21
- swapper, 38-1
- swapping requests, 33-19, 33-25
- swapping, 2-19
- SwapXBusEar, 41-10
- synchronizing execution, 30-16
- syntax errors in file specifications, 12-40
- Sys.keys, 11-33
- SysGen, 43-1
- SysInit.jcl, 33-8
- system configuration block, 27-7
- system configuring
 - number of open system semaphores, 31-6
 - W-, X-, and Z-blocks, 32-5
- system configuration options (See system configuring)
- system date/time format, 26-21
- system date/time structure, 26-21, 27-7
- system events, 29-2
- system initialization, 36-9
- system input process, 11-46 to 11-49
- system keyboard file, 1-5, 11-18 (See also NlsKbd.sys)
- system memory organization, 2-14 to 2-18
- system performance, 3-14
- system profile keyboard, 11-4, 11-41
- system request file, 33-7 (See also Request.sys)
- system requests, 5-5, 33-28
- system semaphores, 31-3, 31-5
- system services
 - built-in, 33-5
 - comparing program model to client, 33-4
 - comparing request-based to system-common, 34-3 to 34-7
 - components, 33-7
 - converting to multi-instance service, 33-21
 - defined, 33-1, 34-1
 - deinstalling, 33-8, 33-26 to 33-27
 - dynamically installable, 33-6
 - execution model, 33-2
 - guidelines for writing, 28-2, 33-8 to 33-11
 - initialization, 33-8
 - multi-instance, 33-2
 - overview of operation, 33-2 to 33-5
 - partition components, 33-11
 - program model, 33-13
 - request-based model, 34-1 to 34-2
 - restrictions and requirements of operation, 33-14
- system service exchange, 33-6

SystemCommonInstall, 34-9, 34-12
SystemCommonQuery, 34-12
system-common NLS table area,
45-17
system-common numbers, 34-10,
39-6
system-common procedures, 4-4
system-common services, 28-3
deinstalling, 34-5, 34-8
guidelines for writing, 34-8 to
34-9
model overview, 34-2

T

tape byte streams (*See* sequential
access byte streams)
target ID, 18-5, 18-7
target keyboard, 11-31
target status, 18-16
task switch, 34-5
Telephone Service configuration
block, 27-7
Telephone Service configuration file
format, 27-7
Telephone status structure, 27-7
terminal output buffer, 27-7
terminal processor, 2-7
TerminatePartitionTasks, 36-16,
36-23
terminating DLL clients, 39-15
terminating programs, 5-5 to 5-5,
36-16
termination requests, 5-5, 33-18,
36-16
text editing, 26-27
TextEdit, 26-27, 26-43
thread of execution, 34-2, 29-1
throughput, 18-5
time slicing, 29-2

timer pseudointerrupt blocks,
27-7, 37-6
timer request blocks, 5-6, 27-8, 37-3
timing single events, 37-4
timing in 100 millisecond intervals,
37-1
toggle, 11-4
token, 12-36
TP (*See* terminal processor)
TP boards, 17-1
TPIB (*See* timer pseudointerrupt
block)
translating, 11-4
translation data block header, 27-8
translation table, 27-8
TransmitCommLineDma, 16-6
trap gate, 40-27
trap handlers, 40-27
traps, 40-4, 40-4
TRB (*See* timer request block)
TruncateDaFile, 23-4
TsVersion, 26-39
typematic key, 11-16
type-ahead buffer, 11-4, 11-9,
11-16, 11-54
discarding contents, 11-17 to
11-17
writing to, 11-17

U

UCB (*See* user control block)
ULCmpB, 26-31, 45-30
underrun, 40-11
unencoded mode, 11-4
unencoded plus, 11-12, 11-13
unencoded value, 11-4
UnlockInContext, 36-21
UnlockVideo, 10-23
UnlockVideoForModify, 10-23

- UnlockXbis, 41-10
 - UnmapBusAddress, 42-6, 42-8
 - UnzoomBox, 26-33
 - UpdateFpMountTable, 33-10
 - UpdateOverlayLru, 38-17
 - updating SRP master processor
 - name table, 33-10
 - user bit, 11-16
 - user control block, 12-34, 27-8
 - user number, 2-11, 2-13, 5-1, 11-16,
 - 28-3, 36-8, 36-14 (*See also* partition)
 - user structure, 11-16, 36-3, 36-5,
 - 36-17
 - UserSysCommonLabel.asm, 34-9,
 - 34-10
 - using byte streams, 8-3 to 8-4
 - using critical section semaphores,
 - 31-9
 - using CTOS operations, 4-3 to 4-4
 - using default response exchanges,
 - 30-18
 - using DMA buffers, 42-6
 - using ENLS operations, 4-15
 - using exchanges to transmit data,
 - 30-4
 - using long-lived memory, 24-10
 - using medium model procedures in DLLs, 39-13
 - using multibyte strings, 11-40
 - using overlays 5-4 (*See also* virtual code management)
 - using parameter management, 6-2
 - using passwords for system access,
 - 12-13
 - using path handles, 18-9
 - using PrintFile operations, 26-16 to 26-17
 - using requests for asynchronous I/O, 30-30
 - using Screen Setup to specify video characteristics, 10-10
 - using SCSI target check or wait operations, 18-23
 - using semaphores, 31-2
 - using short-lived memory, 24-11
 - using system-common procedures in a client, 34-11
 - using termination requests, 33-19
 - using the current screen setup,
 - 10-3
 - using the Module Definition Utility,
 - 39-11
 - using the SCSI manager, 18-23
 - using the stack, 39-14
 - using timer management operations,
 - 37-2
 - using unique workstation hardware IDs, 26-27
 - using VAM
 - for advanced text processing,
 - 10-11
 - for forms-oriented interaction,
 - 10-10
 - using X-bus operations to access module memory, 41-3
 - utility operations, functions of, 26-1
- ## V
- V.35, 2-9, 16-2
 - VacatePartition, 36-16, 36-23
 - validating
 - file specifications, 12-38 to 12-40
 - protection levels, 12-16 to 12-17
 - VAM (*See* Video Access Method)

- variable length parameter block
 - 6-3, 6-4, 24-10, 27-8
 - constructing, 6-8
 - example, 6-6
 - initializing memory, 6-8
 - structure, 6-9
- variable partition memory
 - management, 2-12, 2-14 to 2-16, 3-3, 3-4, 4-13
- variable partition operating
 - systems, 36-1, 36-9, 36-11, 38-1, 38-8
 - multiple CPUs, 42-3
 - single CPU, 42-3
- variable partitions, 1-7, 2-13, 24-10, 36-7
- variable-length records, 20-1
- varying cache buffer size, 25-10
- VCB (See video control block)
- Video Access Method, 10-8
- VDM (See video display management)
- verify mode (See mode 3 DMA)
- VHB (See volume home block)
- video access method, 10-3
- video byte streams, 8-9
 - filtering to a file, 8-4
 - redirecting to a file, 5-4
- video byte streams special
 - characters, 10-5
- video control block, 10-17, 27-8
- video display management, 10-3, 10-8
- video frame 0, 26-16
- virtual 8086 mode, 1-1, 1-6, 3-13
- virtual circuit connection (See SCSI Path)
- virtual code management, 2-22, 24-7, 38-1
 - and the Linker, 38-4, 38-14
 - and virtual memory operating systems, 38-1
 - call/return conventions, 38-13
 - data structures, 38-4 to 38-7
 - model overview, 38-3 to 38-3
 - protected mode operation, 38-8 to 38-9, 38-15
 - real mode operation, 38-9 to 38-13, 38-14, 38-15
 - retrofitting overlay programs, 38-2
 - stack frame, 38-10
 - tracing the stack, 38-11 to 38-13
- virtual code management data
 - structures
 - overlay descriptors, 38-6
 - overlay zone header, 38-6
 - ProcInfoNonres, 38-7
 - ProcInfoRes, 38-6
 - return overlay descriptors, 38-7
 - StaticsDesc, 38-6
 - stubs array, 38-6
- virtual memory management,
 - xxxix, xl, 2-13, 2-16, 3-3, 4-13
- virtual memory operating systems,
 - 36-1, 36-9, 38-1, 40-28, 42-1
- virtual memory, xxxvii, 3-3
- VLPB (See variable length parameter block)
- Voice control structure, 27-8
- Voice file header, 27-8
- Voice file record, 27-8
- Voice Processor control block, 27-8

Voice/Data operations

- AsGetVolume, 35-14
- TsConnect, 35-14
- TsDataChangeParams, 35-14
- TsDataCheckpoint, 35-14
- TsDataCloseLine, 35-14
- TsDataOpenLine, 35-14
- TsDataRead, 35-15
- TsDataRetrieveParams, 35-15
- TsDataUnacceptCall, 35-15
- TsDataWrite, 35-15
- TsDeinstall, 35-15
- TsDial, 35-15
- TsDoFunction, 35-15
- TsGetStatus, 35-15
- TsHold, 35-15
- TsOffHook, 35-15
- TsOnHook, 35-16
- TsQueryConfigParams, 35-16
- TsReadTouchTone, 35-16
- TsRing, 35-16
- TsSetConfigParams, 35-16
- TsVoiceConnect, 35-16
- TsVoicePlaybackFromFile, 35-16
- TsVoiceRecordToFile, 35-16
- TsVoiceStop, 35-16

volatile memory (*See* memory addresses)

volume control structures, 13-4

- allocation bit map, 12-32
- bad sector file, 12-32
- device control block, 12-35
- directories, 12-33
- disk extent, 12-6, 12-32
- extension file header block, 12-32
- file area block, 12-34
- file control block, 12-34
- file header blocks, 12-6, 12-30, 12-32

I/O block, 12-34

- master file directory, 12-6, 12-33
- system directory, 12-33
- user control block, 12-34
- volume home block, 12-6, 12-30, 12-34

volume control structures,
duplication of, 2-5

volume encryption, 12-18

volume home block, 12-6, 12-11,
12-23, 12-34, 27-8

volume home block, accessing, 13-4

volume name, 13-2

volume passwords, 12-8, 12-12

W

wait exchange table, 32-7

wait flags table, 32-7

Wait, 30-16, 30-53, 30-54

waiting for several semaphores to
clear, 31-13

waiting state, 29-5

WaitLong, 30-54

wait-satisfied flags table, 32-7

wild card operations, 12-41

WildCardClose, 12-44

WildCardInit, 12-41, 12-44

WildCardMatch, 26-15, 26-32

WildCardNext, 12-41, 12-44

working set, 3-3, 3-9

workstation agent, 30-33, 30-34,
30-51

workstation operating system
features, 2-8

workstation video capabilities,
10-11 to 10-16

- workstations
 - server, 2-7
 - local file system, 2-7
 - diskless cluster workstations, 2-7
 - EISA/ISA-bus, 13-4
 - SuperGen Series 5000, 13-4
- writable segments, 11-26
- Write, 12-43
- WriteAsync, 12-49, 30-30
- WriteBsRecord, 8-15
- WriteByte, 8-15
- WriteByteStreamParameterC,
15-7, 16-3
- WriteCommLineStatus, 16-4, 16-5,
40-32
- WriteDaFragment, 23-2, 23-5
- WriteDaRecord, 23-4
- WriteHardId, 26-27, 26-43
- WriteIBusDevice, 11-55, 11-59
- WriteIBusEvent, 11-55, 11-59
- WriteKbdBuffer, 11-17, 11-58
- WriteLog, 26-28, 26-43
- WriteRsRecord, 22-3
- WriteStatusC, 15-7, 16-4
- write-back, 25-2
- write-behind mode, 23-3
- write-through mode, 23-3
- write-through, 25-2
- writing a CTOS call, 4-3 to 4-4
- writing device-independent
programs, 7-3
- writing language statements, 4-3
- writing pages to backing store,
3-11
- writing records to the system log
file, 26-28
- writing to the type-ahead buffer,
11-17
- writing video programs for different
workstation models, 10-16
- W-block, 32-2, 32-5, 32-6
- X**
 - X.21, 2-9
 - X.25 byte streams, 8-9
 - X.25, 16-2
 - XBIF, 41-7
 - XbifVersion, 26-39
 - XBIS (See X-Bus initialization
structure)
 - XC002Version, 26-39
 - X-Bus
 - assigning I/O addresses to X-Bus
modules, 41-1
 - DMA, 41-6
 - initialization structure, 41-6
 - interrupts, 41-8
 - management, 41-1 to 41-8
 - X-Bus initialization structure, 41-6
 - X-Bus modules
 - accessing in protected mode, 41-5
 - accessing in real mode, 41-5
 - accessing module memory, 41-3
 - communicating with the processor
module, 41-6
 - memory usage classes, 41-3
 - X-Bus interrupt handlers, 40-23
 - X-Bus+, 41-1
- Y**
 - Y-block, 32-2, 32-5, 32-6
- Z**
 - Z-block, 32-2, 32-5, 32-6
 - ZoomBox, 26-16, 26-33
 - ZPrint, 26-17, 26-34
 - [VID]0, 26-16



43602630-000